

T.C.
SAKARYA ÜNİVERSİTESİ
BİLGİSAYAR VE BİLİŞİM BİLİMLERİ FAKÜLTESİ

BSM 498 BİTİRME ÇALIŞMASI

KRIPTO CÜZDANI

**G171210009 - Arif DAMAR
G171210028 - Ömer Çağrı ŞAYİR**

**Fakülte Anabilim Dalı : BİLGİSAYAR MÜHENDİSLİĞİ
Tez Danışmanı : Öğr.Gör.Dr. Yüksel YURTAY**

2021-2022 Bahar Dönemi

ÖNSÖZ

Günümüzde kripto varlıklara olan ilgi hızla artmaktadır ve bu artan ilgi karşısında da sürekli yeni borsalar bu piyasaya girmektedir. Bazı kripto varlıkların sadece bazı borsalarda olması sebebiyle kullanıcıların birden fazla borsayı aynı anda kullanması gerekmektedir. Bu durumun sonucu olarak da kullanıcılar, hangi borsada ne kadar kripto varlığına sahip olduğunu takip etmekte zorlanmaktadır. Projemiz, kullanıcıların farklı borsalarda bulunan varlıklarını tek bir arayüz üzerinden takip etmeye olanak sağlamayı amaçlamaktadır.

Başta Tez Danışmanımız Dr. Yüksel Yurtay'a olmak üzere, ücretsiz API'ler sağlayarak bu projenin var olabilmesinin önünü açan büyük Kripto Borsalarına, YouTube üzerinden yayinallyı ücretsiz Binance API eğitim videolarıyla bize teknik çözümler sunan [Part Time Larry](#) kanalına ve tüm bu süreç boyunca bizlere gerek teknik anlamda gerek manevi anlamda destek olan tüm arkadaş ve ailemize teşekkürlerimizi iletiyoruz.

İÇİNDEKİLER

ÖNSÖZ	2
İÇİNDEKİLER	3
SİMGELER VE KISALTMALAR LİSTESİ	6
ÖZET	7
GİRİŞ	1
TEKNOLOJİLERİN İNCELENMESİ	3
Sunucu Katmanı	3
Node.js	4
Express.js	4
TypeScript	4
Mongoose	4
Job Handler Katmanı	5
Node.js	5
TypeScript	5
Cron Expression	5
Mongoose	5
Arayüz Katmanı	6
React.js	6
Tailwind CSS	6
React Router	7
Context	7
Mantine UI Kütüphanesi	7
Veri Tabanı	7
Veri tabanı şeması	8
Mongoose	9
BACKEND (EXPRESS.JS API)	10
Veritabanı Modelleri	10

User (Kullanıcı) Modeli	10
UserCrypto Modeli	10
Wallet Modeli	10
Exchange Modeli	10
ChangeRecord Modeli	11
RecurringJob Modeli	11
Routing	11
Auth Endpointleri	12
Register	14
Login	14
Logout	16
Me	17
Wallet Endpointleri	18
Get	19
Dashboard	19
Exchange Endpointleri	21
Create	22
Refresh	23
List	23
Get One	24
UserCrypto Endpointleri	25
Add	26
Update	27
Delete	28
JOB HANDLER SERVİSİ	30
Veritabanı Modelleri	30
RecurringJob Modeli	30
ChangeRecord Modeli	30
İşleyiş	31
FRONTEND (REACT.JS)	34

Register (Kayıt Ol) Sayfası	34
Login (Giriş Yap) Sayfası	36
Navbar Yapısı	37
Main Sayfası	38
Balance Component	39
User's Assets Component	40
SingleCrypto Component	41
Günlük Değişim Grafiği	43
Yeni Kripto Ekleme Modali	44
Sayfa Yönlendirme	45
UYGULAMADAN EKRAN GÖRÜNTÜLERİ	46
Giriş Ekranı	46
Kayıt Ekranı	48
Ana Sayfa	49
SONUÇLAR	56
KAYNAKLAR	57

SİMGELER VE KISALTMALAR LİSTESİ

BTC	: Bitcoin
ETH	: Ethereum
JS	: JavaScript
TS	: TypeScript
CRUD	: Create, Read, Update, Delete

ÖZET

Anahtar kelimeler: Blockchain (Blok zincir), Cripto Varlıklar, Cripto Borsaları, Web Teknolojileri

Cripto para, tamamen dijital, şifrelenmiş, sanal para birimidir. Cripto para birimlerine sahip olmanın yöntemi reel para birimleri (Türk Lirası, Dolar, Euro vb.) ile satın almaktır. Bu satın alma işleminin gerçekleştirildiği sitelere "Cripto Para Borsası" denmektedir.

Cripto Cüzdanı projesi, kullanıcıların üyesi olduğu farklı Cripto Borsalarındaki farklı Cripto Varlıklarını tek bir uygulama/arayüz üzerinden takip edebilme ihtiyacına çözüm olmayı hedeflemiştir. Bu uygulama sayesinde birden fazla Cripto Borsa kullanan kişiler, bütün Cripto Varlıklarını tek arayüz üzerinden takip edebilmektedir.

Bu bitirme çalışmasında, kullanıcıların Cripto Varlıklarını takip edebileceği en büyük Cripto Borsalarından olan Binance ve KuCoin borsaları bulunmaktadır. Kullanıcılar, arayüz üzerinden istedikleri borsayı seçerek o borsada bulunan Cripto Varlıklarını görüntüleyebilir ve kendi cüzdanlarına istedikleri miktarda Cripto Varlık ekleyerek takibini sağlayabilmektedirler

BÖLÜM 1. GİRİŞ

Projenin GitHub Linki: <https://github.com/cagrisayir/advanced-coin-wallet>

Geliştirilen API'nin dökümantasyon linki:

<https://documenter.getpostman.com/view/18572789/UVRHhiS9>

Bu projede kripto varlık sahibi kullanıcıların varlıklarının bulunduğu farklı borsalardaki varlıklarını tek bir uygulamada takip etmeleri amaçlanmıştır. Bu uygulama sayesinde tek bir arayüzde borsa fark etmeksiz tüm kripto varlıkların anlık durumu ve kâr/zarar oranı takip edilebilmektedir.

Bu bitirme projesinde, kullanıcıların Kripto Varlıklarının bulunabileceği en büyük Kripto Varlık Borsalarından ikisi olan Binance ve KuCoin borsaları bulunmaktadır. Uygulamayı kullanacak olan kullanıcılar bu iki borsada olan tüm varlıkları, uygulama içerisindeki Cüzdanlarına ekleyebilir ve Ana Sayfa üzerinden Cüzdanlarının zamana bağlı değişimini takip edebilirler.

1.1. Kullanılan Teknolojiler

Projede kullanılan teknolojiler şu şekildedir;

- **Backend**

- Node.js
- Express.js
- TypeScript
- Session Authentication
- Mongoose

- **Job Handler Servisi**

- Node.js
- Cron Expressionları
- TypeScript
- Mongoose

- **Frontend**

- Node.js
- React.js
- Tailwind CSS
- TypeScript
- Mantine
- React-Router
- Context API

- **Database**

- NoSQL
- MongoDB

BÖLÜM 2. TEKNOLOJİLERİN İNCELENMESİ

Proje; backend, job handler, frontend ve veri tabanı olmak üzere 4 servisten oluşmaktadır. Backend katmanı, Node.js çalışma ortamında Express.js sunucu çatısı ve TypeScript dili ile geliştirilmiştir.

Job handler katmanı, Node.js çalışma ortamında TypeScript dili ile geliştirilmiştir. İşleri belirli aralıklarla çalıştmak için Cron Expressionları kullanır.

Frontend katmanı, Node.js çalışma ortamında React.js kütüphanesi ve TypeScript dili ile geliştirilmiştir.

Veri tabanı için ise bir NoSQL veri tabanı olan MongoDB kullanılmıştır.

2.1. Sunucu Katmanı

Backend katmanı, Node.js çalışma ortamında Express.js sunucu çatısı ve TypeScript dili ile geliştirilmiştir.

2.1.1. Node.js

- JavaScript'i sunucu katmanında kullanabilmeyi sağlayan bir çalışma ortamıdır.
- Temelinde JavaScript ile çalıştığı için çok dinamik ve hızlıdır.

2.1.2. Express.js

- Express.js web sunucu çatısı, backend projesinin bir API olarak hizmet vermesini sağlamak amacıyla kullanılmaktadır.
- API'ye gelen isteklerin yönetilmesini sağlamaktadır. Farklı tipteki isteklere (GET, POST, PUT, PATCH, DELETE vs.) uygun olacak şekilde yönlendirmeler yapılmasını sağlamaktadır.
- Gelen isteklerin ilgili endpointlere ilettilmesini sağlamaktadır.
- Middleware'ler sayesinde bir istek pipeline'ının her noktasına erişilebilmesini sağlamaktadır.

2.1.3. TypeScript

- Uygulamaları geliştirme esnasında JavaScript'in eksik bir yanı olan statik veri tiplerine sahip, nesne yönelimli ve derlenebilir bir programlama dilidir.
- Geliştirme sürecinde yapılabilecek hataları minimuma indirmeyi sağlar.
- Derlenme sonrası yazılan bütün TypeScript kodu JavaScript koduna çevrilir ve bu JavaScript kodu çalıştırılır. Bu sebeple JavaScript'in çalışabildiği her yerde TypeScript yazılabilir.

2.1.4. Mongoose

- NoSQL bir veri tabanı olan MongoDB ile Node.js projesinin iletişim kurabilmesini sağlayan bir ODM (Object Document Mapping) çözümüdür.
- Verileri modellemek için Schema adındaki sınıflar kullanılır. Bunlar, veri şemaları oluşturabilmek için kullanılır.

2.2. Job Handler Katmanı

Job Handler katmanı, Node.js çalışma ortamında TypeScript ile geliştirilmiştir. İşlerin sıraya alınmasıyla ilgili Cron Expressionlardan faydalانılmıştır.

2.2.1. Node.js

- JavaScript'i sunucu katmanında kullanabilmeyi sağlayan bir çalışma ortamıdır.
- Temelinde JavaScript ile çalıştığı için çok dinamik ve hızlıdır.

2.2.2. TypeScript

- Uygulamaları geliştirme esnasında JavaScript'in eksik bir yanısı olan statik veri tiplemelerine sahip, nesne yönelimli ve derlenebilir bir programlama dilidir.
- Geliştirme sürecinde yapılabilecek hataları minimuma indirmeyi sağlar.
- Derlenme sonrası yazılan bütün TypeScript kodu JavaScript koduna çevrilir ve bu JavaScript kodu çalıştırılır. Bu sebeple JavaScript'in çalışabildiği her yerde TypeScript yazılabilir.

2.2.3. Cron Expression

- Boşluklar ile ayrılan 6 alana sahip bir string ifadedir.
- “* * * * *” şeklinde ifade edilir.
- Sırasıyla saniye (0-59), dakika (0-59), saat (0-23), ayın günü (1-31), ay (1-12), haftanın günü (0-7) şeklinde anlatılır.

2.2.4. Mongoose

- NoSQL bir veri tabanı olan MongoDB ile Node.js projesinin iletişim kurabilmesini sağlayan bir ODM (Object Document Mapping) çözümüdür.
- Verileri modellemek için Schema adındaki sınıflar kullanılır. Bunlar, veri şemaları oluşturabilmek için kullanılır.

2.3. Arayüz Katmanı

Arayüz katmanı, Node.js çalışma ortamında React.js kütüphanesi ve TypeScript dili ile geliştirilmiştir.

2.3.1. React.js

Projede Frontend Kütüphanesi olarak React kullanılmıştır. React, Single Page Application yapmamızı sağlayan bir kütüphanedir. Bu projenin client tarafında Sayfaların olduğu Pages klasörü, yazılım parçalarının olduğu components klasörü, server'dan bilgilerin çekildiği api klasörü vardır. Özellikle Pages ve components klasörlerinin içinde kullanılan .tsx dosyaları React ile yazılmış olup, herhangi bir değişiklikti sadece belirli component'in yenilenmesi sağlanmıştır. Örneğin Header ve Sidebar componentları oluşturulmuştur ve bu sayede bu kod parçaları birden fazla sayfada kullanılmıştır.

2.3.2. Tailwind CSS

Projede tasarım yapmak için CSS framework'ü olan Tailwind CSS kullanılmıştır. Tailwind CSS sayesinde her html tag'ının içinde className parametresi içerisinde verdiğimiz değerler ile (örneğin 'flex' ile display özelliğinin flex olmasını sağlarız) sayfaların stillendirilmesi sağlanmıştır. Tailwind CSS arka planda tüm sınıfların tanımlandığı ve geliştiricinin kullanmak istediği stile ve özelliğe göre önceden tanımlanmış sınıfları kullandığı bir framework'tür.

2.3.3. React Router

Projenin ön yüzünde sayfa ve link geçişleri için React Router kütüphanesi kullanılmıştır. Bu kütüphane React ile birlikte çalışmakta olup linkler arasında geçiş yapmaya olanak sağlamaktadır.

2.3.4. Context

Bu projede state management için React Context kullanılmıştır. Context, proje içinde birden fazla component içinde kullanılan veriler için kullanılmıştır. Context ile proje içinde veri erişiminin tek bir yerden yönetilmesi sağlanmıştır.

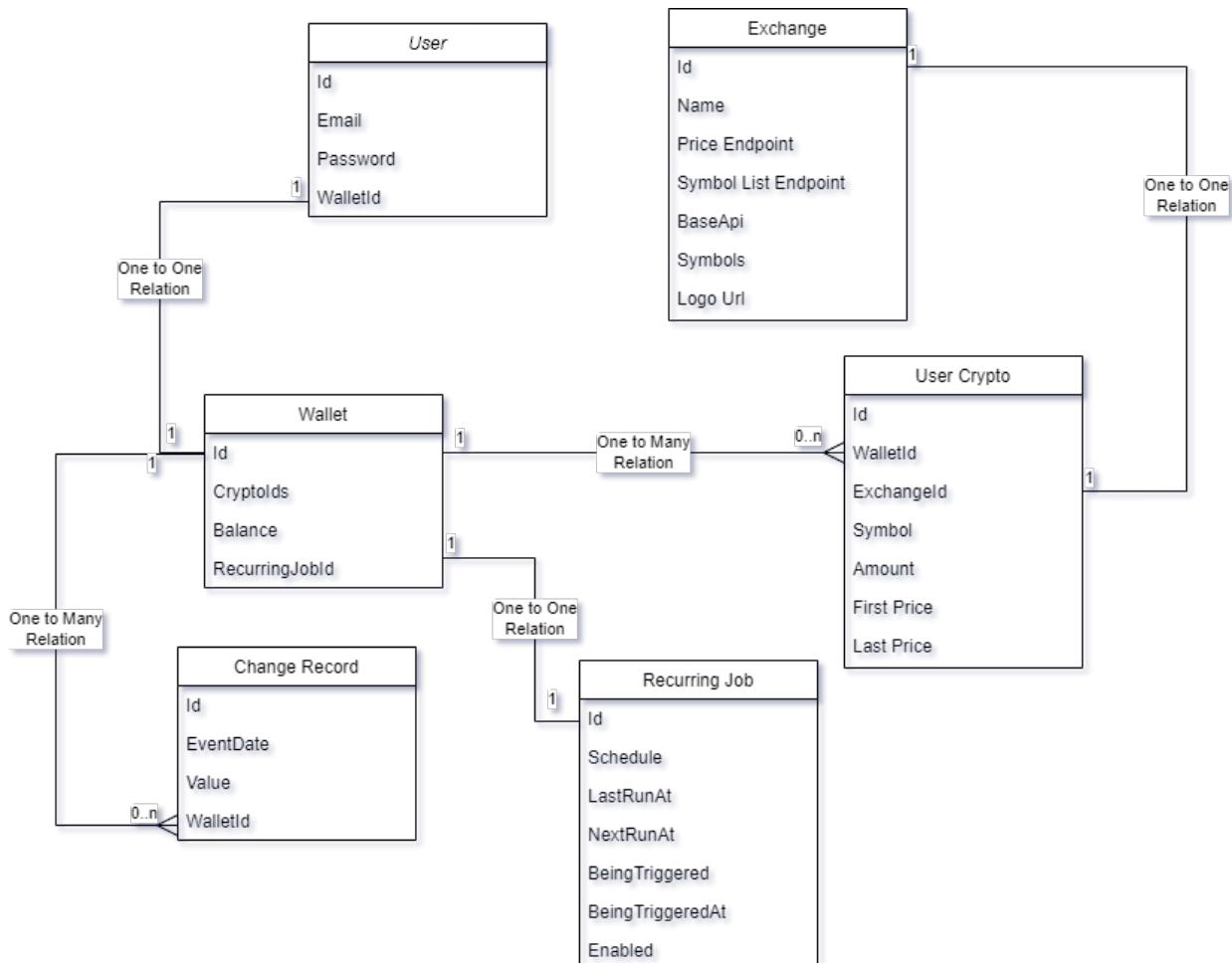
2.3.5. Mantine UI Kütüphanesi

Mantine bir UI kütüphanesi olup, projenin bazı yerlerinde bu kütüphanenin sağladığı hooklar ve tasarımlar kullanılmıştır.

2.4. Veri Tabanı

Projede veri tabanı olarak NoSQL bir veri tabanı çözümü olan MongoDB kullanılmıştır. Dizi tipinde alanlar tutabilmesi, projenin ihtiyaçlarına uygun olacak şekilde hızlı ve dinamik bir veri tabanı olduğu için tercih edilmiştir.

2.4.1. Veri tabanı şeması



Şekil 2.4.1. Veri Tabanı UML Şeması

UML şemasından da görülebileceği üzere, projede 4 adet tablo bulunmaktadır. Bu tablolar arasındaki ilişkiler şu şekildedir;

- User ile Wallet arasında 1-1 bir ilişki bulunmaktadır.
- Wallet ile User Crypto arasında 1-N bir ilişki bulunmaktadır.
 - NoSQL bir veri tabanı olan MongoDB kullanıldığı için User Crypto tablosunda WalletId bulunmasının yanı sıra Wallet tablosunda da bir dizi olarak CryptoIds alanı bulunmaktadır. Bu alanın bulunması, Wallet'ın detaylarını görüntüleme esnasında yapılan Population işlemini kolaylaştırmaktadır.
- Wallet ile Recurring Job arasında 1-1 bir ilişki bulunmaktadır.
- Wallet ile Change Record arasında 1-N bir ilişki bulunmaktadır..

- User Crypto ile Exchange arasında 1-1 bir ilişki bulunmaktadır.

2.4.2. Mongoose

- Mongoose, NoSQL bir veri tabanı olan MongoDB ile Node.js projesinin iletişim kurabilmesini sağlayan bir ODM (Object Document Mapping) çözümüdür.
- Verileri modellemek için Schema adındaki sınıflar kullanılır. Bunlar, veri şemaları oluşturabilmek için kullanılır.
- Şemalar oluştururken sunduğu interface desteği sayesinde TypeScript dili ile son derece uyumludur.
- CRUD operasyonlarını kolaylaştırmasının yanı sıra oldukça kompleks sorgular yapmaya ve yüksek performansta population yapmaktadır.

BÖLÜM 3. BACKEND (EXPRESS.JS API)

Backend geliştirmesi TypeScript. MongoDB, JOI kütüphanesi, NodeJS kullanılarak yapılmıştır.

3.1. Veritabanı Modelleri

Veritabanı için şu ana kadar MongoDB kullanıldı. MongoDB'nin Bulut Tabanlı uygulaması MongoDB Atlas üzerinde çalışıldı.

Projede var olan veritabanı modelleri ve bu modellerdeki alanlar şunlardır:

- User (Kullanıcı) Modeli
 - email
 - password
 - walletId
 - fullName
- UserCrypto Modeli
 - walletId
 - exchangeId
 - symbol
 - amount
 - firstPrice
 - lastPrice
- Wallet Modeli
 - cryptoIds
 - balance
 - recurringJobId
- Exchange Modeli
 - name
 - baseApi
 - symbolListEndpoint
 - priceEndpoint
 - symbols
 - logoUrl

- ChangeRecord Modeli
 - eventDate
 - value
 - walletId

- RecurringJob Modeli
 - schedule // cron expression
 - lastRunAt
 - nextRunAt
 - beingTriggered
 - beingTriggeredAt
 - enabled

3.2. Routing

API'ın routing yapılandırması yapılırken öncelikle tüm routeler `/api` altına alınmıştır. Sonrasında endpointler, yaptıkları işlere göre mantıksal olarak farklı routeler altına toplanmıştır. Projede tanımlı olan tüm routeler aşağıdaki şekilde verilmiştir.



```

import express from "express";
import authRoutes from "./auth";
import walletRoutes from "./wallet";
import exchangeRoutes from "./exchange";
import userCryptoRoutes from "./userCrypto";
import { SuccessResult } from "../../models/result";

const router = express.Router();

router.get("/", (_req: express.Request, res: express.Response) => {
  return res
    .status(200)
    .json(new SuccessResult("API is up and running!", null));
});

router.use("/auth", authRoutes);
router.use("/wallet", walletRoutes);
router.use("/exchange", exchangeRoutes);
router.use("/userCrypto", userCryptoRoutes);

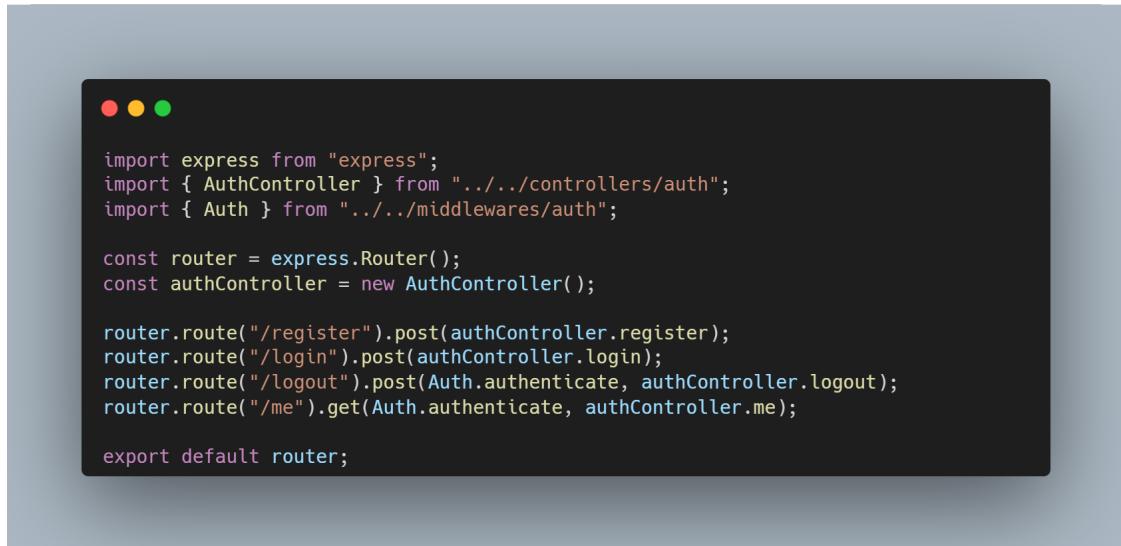
export default router;

```

Şekil 3.2.1.. Auth endpointlerini

Auth Endpointleri

Authentication için Register (kayıt olma), Login (giriş yapma), Logout (çıkış yapma) ve o an giriş yapmış kullanıcının bilgilerinin bulunduğu Me adında dört adet endpoint tanımlanmıştır.



```
● ● ●

import express from "express";
import { AuthController } from "../../controllers/auth";
import { Auth } from "../../middlewares/auth";

const router = express.Router();
const authController = new AuthController();

router.route("/register").post(authController.register);
router.route("/login").post(authController.login);
router.route("/logout").post(Auth.authenticate, authController.logout);
router.route("/me").get(Auth.authenticate, authController.me);

export default router;
```

Şekil 3.2.2. Auth endpointlerinin belirtildiği kod tanımı

Register ve Login endpointlerine her kullanıcı erişebilirken, Logout ve Me endpointlerine sadece giriş yapmış kullanıcıların erişebilmesi Auth sınıfının authenticate Middleware'sı ile sağlanmıştır.

```
● ● ●  
1 export class Auth {  
2     public static async authenticate(request: Request, response: Response, next: NextFunction) {  
3         try {  
4             const userId = request.session.userId;  
5  
6             if (userId) {  
7  
8                 const user = await User.findById(userId);  
9  
10                if (!user) {  
11                    return response.status(404).send(new FailureResult('User not found'));  
12                }  
13  
14                response.locals.user = user;  
15                return next();  
16            } else {  
17                return response.status(401).json(new FailureResult('Unauthenticated'));  
18            }  
19        } catch (error) {  
20            console.log(error);  
21        }  
22    }  
23 }  
24 }
```

Şekil 3.2.3. Auth.authenticate middleware'sının belirtildiği kod tanımı

Register

Register (kayıt olma) için kullanıcıdan gelen e-mail ve şifre tutulmuştur. Önce bu veriler JOI kütüphanesi yardımcı ile önceden belirlenen kurallara uyup uymadığı kontrol edilmiştir (validation). Daha sonra veritabanında aynı e-mail olup olmadığı kontrol edilmiştir. Eğer bu e-mailde bir kullanıcı bulunduğu话sa “Bu e-mailde bir kullanıcı zaten mevcuttur. Lütfen başka e-mail ile kayıt olmayı deneyiniz.” uyarısı gönderilmiştir.

Eğer böyle bir e-mail mevcut değilse önce 0 balance ile bir Wallet (cüzdan) oluşturuluyor. Daha sonra kullanıcının şifresi kriptoloji ile alakalı kütüphaneler ile (bcrypt) şifrelenip, e-mail, şifreli parola ve yeni oluşturulan cüzdan ID'si ile beraber yeni bir kullanıcı oluşturuluyor.

```

1  public async register(request: Request, response: Response) {
2    try {
3      const { email, password } = request.body;
4
5      await userRegisterValidator.validateAsync({ email, password });
6
7      const existingUser = await User.findOne({ email });
8
9      if (existingUser) {
10        return response.status(400).send(new FailureResult("A user with this email already exists. Try signing in."));
11      }
12
13      const userWallet = await Wallet.create({ balance: 0 });
14
15      const hashedPassword = await bcrypt.hash(password, 12);
16      const result = await new User({
17        email,
18        password: hashedPassword,
19        walletId: userWallet.id,
20      }).save();
21
22      if (result != null) {
23        return response.status(201).send(new SuccessResult("Registered successfully", null));
24      }
25
26      return response.status(500).send(new SuccessResult("Registration failed", null));
27    } catch (error: any) {
28      if (error.isJoi) {
29        return response.status(400).send(new FailureResult("Validation error: " + error.message));
30      }
31
32      console.log(error);
33      return response.status(500).send(new FailureResult("Something went wrong."));
34    }
35  }

```

Sekil 3.2.4.. Register endpointinin kod tanımı

Login

Login için kullanıcıdan gelen e-mail ve şifre öncelikle JOI kütüphanesi yardımı ile doğrulanıyor. Daha sonra bu e-mail'de bir kullanıcı var mı diye Veritabanına bakılıyor. Eğer bu verilerde bir kullanıcı yok ise “Kullanıcı bulunamadı” şeklinde hata dönülüyor.

E-mail var ise bu kullanıcının parolası ile kullanıcının girdiği parola karşılaştırılıyor. Yanlışsa “Girilen bilgiler hatalı” şeklinde bir hata dönülüyor. Doğruysa “Giriş başarılı” şeklinde 200 kodlu bir cevap gönderilir.

```

1  public async login(request: Request, response: Response) {
2    try {
3      const { email, password } = request.body;
4
5      await userLoginValidator.validateAsync({ email, password });
6
7      const existingUser = await User.findOne({ email });
8
9      if (!existingUser) {
10        return response.status(401).json(new FailureResult("User not found"));
11      }
12
13      const isPasswordCorrect = await bcrypt.compare(password, existingUser.password);
14
15      if (!isPasswordCorrect) {
16        return response.status(401).json(new FailureResult("Invalid credentials"));
17      }
18
19      request.session.userId = existingUser._id.toString();
20
21      return response.status(200).json(new SuccessResult("Login successful", null));
22    } catch (error: any) {
23      if (error.isJoi) {
24        return response.status(400).send(new FailureResult("Validation error: " + error.message));
25      }
26
27      console.log(error);
28      return response.status(500).json(new FailureResult("Something went wrong."));
29    }
30  }

```

Şekil 3.2.5. Login endpointinin kod tanımı

Logout

Logout işleminde authenticate olmuş kullanıcının session'ı destroy metodu ile silinir. Silme işlemi sırasında sunucu taraflı bir hata olması durumunda kullanıcıya ilgili hata mesajı gönderilir. Hata olmaz ise kullanıcının session'ı silinir ve 204 status code'u dönülür.

```
public async logout(request: Request, response: Response) {
  request.session.destroy((err) => {
    if (err) {
      console.log(err);
      return response.status(500).send(new FailureResult("Something went wrong."));
    }

    return response.status(204).send();
  });
}
```

Şekil 3.2.6. Logout endpointinin kod tanımı

Me

Me işleminde authenticate olmuş olan kullanıcının email ve fullName bilgileri 200 status code'u ile birlikte kullanıcıya dönülür.

```
public async me(_request: Request, response: Response) {
  try {
    const currentUser: IUser = response.locals.user;

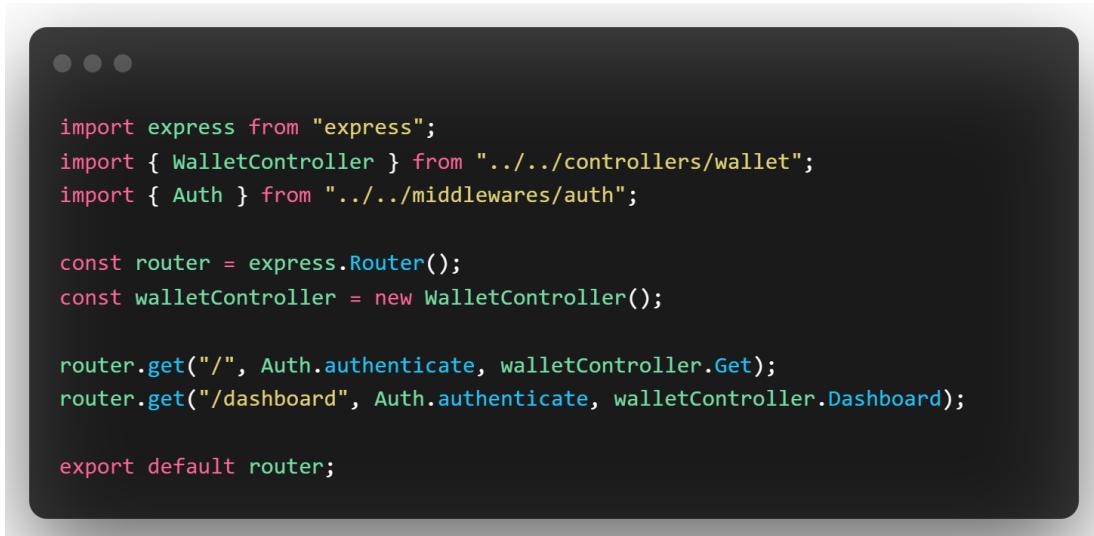
    const userMeDto = (({ email, fullName }) => ({ email, fullName }))(currentUser);

    return response.status(200).send(new SuccessResult("Current user fetched successfully",
userMeDto));
  } catch (error) {
    console.log(error);
    return response.status(500).send(new FailureResult("Something went wrong."));
  }
}
```

Şekil 3.2.7. Logout endpointinin kod tanımı

Wallet Endpointleri

Wallet için kullanıcının cüzdanını, detaylarıyla (mevcut kripto varlıklarını ve cüzdanın toplam değeri) birlikte getiren Get endpointi ve cüzdanının toplam değerinin zamana göre değişimini getiren Dashboard endpointi olmak üzere 2 adet endpoint tanımlanmıştır.



```
import express from "express";
import { WalletController } from "../../controllers/wallet";
import { Auth } from "../../middlewares/auth";

const router = express.Router();
const walletController = new WalletController();

router.get("/", Auth.authenticate, walletController.Get);
router.get("/dashboard", Auth.authenticate, walletController.Dashboard);

export default router;
```

Şekil 3.2.8. Wallet endpointlerinin belirtildiği kod tanımı

Get

Get işlemi için öncelikle authenticate olmuş olan kullanıcının walletId alanı üzerinden kullanıcıya ait olan Wallet (cüzdan) veri tabanından alınır. Sonrasında ilgili cüzdana bulunan kripto varlıklar veri tabanından alınır. Alınan bu kripto varlıkların güncel değerleri hesaplanır ve cüzdanın toplam değeri hesaplanır. Böylece kullanıcı cüzdanına girdiğinde her zaman en güncel değeri görmesi sağlanmış olur.

```
public async Get(request: Request, response: Response) {
  try {
    const wallet = await Wallet.findById(response.locals.user.walletId);

    if (!wallet) {
      return response.status(404).send(new FailureResult("User wallet not found!"));
    }

    const userCryptos = await UserCrypto.find({ walletId: wallet.id }).populate({
      path: "exchange",
      select: "-symbols -createdAt -updatedAt",
    });

    wallet.balance = 0;
    for (const userCrypto of userCryptos) {
      const currentPrice = await CryptoPriceHelper.getPrice(
        userCrypto.exchange.baseApi + userCrypto.exchange.priceEndpoint + userCrypto.symbol,
        userCrypto.exchange.name
      );

      wallet.balance += +(currentPrice * userCrypto.amount).toFixed(4);
      wallet.balance = +wallet.balance.toFixed(4);
      userCrypto.lastPrice = currentPrice;
      userCrypto.save();
    }

    await wallet.save();
    wallet.cryptos = userCryptos;

    return response.status(200).send(new SuccessResult("Wallet Fetched Successfully!", wallet));
  } catch (error) {
    console.log(error);
    return response.status(500).json(new FailureResult("Something went wrong."));
  }
}
```

Şekil 3.2.9. Get endpointinin kod tanımı

Dashboard

Dashboard işlemi için Job Handler servisinin üretmiş olduğu ChangeRecord kayıtları, authenticate olmuş kullanıcının walletId'si üzerinden alınarak kullanıcıya ilettilir. Bu veri ile kullanıcı, cüzdanının toplam değerindeki değişimi zamana bağlı olarak görüntüleyebilir.

```
public async Dashboard(request: Request, response: Response) {
  try {
    const lastChanges = await ChangeRecordModel.find({ walletId: response.locals.user.walletId })
      .sort({ eventDate: -1 })
      .limit(300);

    return response.status(200).send(new SuccessResult("Dashboard Fetched Successfully!",
lastChanges));
  } catch (error) {
    console.log(error);
    return response.status(500).json(new FailureResult("Something went wrong."));
  }
}
```

Şekil 3.2.10. Dashboard endpointinin kod tanımı

Exchange Endpointleri

Exchange için yeni bir Exchange (Kripto Borsası) oluşturmak için kullanılacak olan create endpointi, veri tabanında kayıtlı olan kripto borsalarındaki sembollerin güncellenmesini sağlayacak refresh endpointi, veri tabanında kayıtlı olan kripto borsalarını semboller ile birlikte listeleyecek olan list endpointi ve parametre olarak verilen id bilgisine göre tek bir kripto borsayı getiren endpoint olmak üzere 4 adet endpoint tanımlanmıştır.

```
import express from "express";
import { ExchangeController } from "../../controllers/exchange";
import { Auth } from "../../middlewares/auth";

const router = express.Router();
const exchangeController = new ExchangeController();

router.post("/create", Auth.authenticate, exchangeController.Create);
router.post("/refresh", Auth.authenticate, exchangeController.RefreshSymbols);
router.get("/", Auth.authenticate, exchangeController.List);
router.get("/:exchangeId", Auth.authenticate, exchangeController.GetOne);

export default router;
```

Şekil 3.2.11. Exchange endpointlerinin belirtildiği kod tanımı

Create

Create işleminde öncelikle *name*, *baseApi*, *symbolListEndpoint*, *priceEndpoint*, *logoUrl* verileri body'den alınır. Alınan bu veriler öncelikle belirtilen kurallar dahilinde validasyona sokulur. Eğer uygun veriler iletilmiş ise aynı isimde veri tabanında kayıtlı olan başka bir kripto borsası olup olmadığı kontrol edilir. Eğer yoksa gönderilen veriler ile yeni bir Exchange oluşturulur.

```
public async Create(request: Request, response: Response) {
  try {
    const { name, baseApi, symbolListEndpoint, priceEndpoint, logoUrl } = request.body;

    await exchangeCreateValidator.validateAsync({ name, baseApi, symbolListEndpoint, priceEndpoint, logoUrl });

    if (await Exchange.exists({ name })) {
      return response.status(400).send(new FailureResult("Exchange already exists."));
    }

    const exchange = await Exchange.create({
      name,
      baseApi,
      symbolListEndpoint,
      priceEndpoint,
      logoUrl,
    });

    return response.status(200).send(new SuccessResult("Exchange Created Successfully!",
exchange));
  } catch (error: any) {
    if (error.isJoi) {
      return response.status(400).send(new FailureResult("Validation error: " + error.message));
    }

    console.log(error);
    return response.status(500).send(new FailureResult("Something went wrong."));
  }
}
```

Şekil 3.2.12. Create endpointinin kod tanımı

Refresh

Refresh işleminde öncelikle *name* verisi body'den alınır. Alınan bu name değerine göre veri tabanından ilgili kripto borsa alınır. Alınan bu kripto borsasının symbol listeleme endpointine istek atılır. Sonrasında farklı borsaların symbol listelerinden farklı yapıda sonuçlar döndürülerek her borsa için ayrı bir parse işlemi uygulanır ve ilgili kripto borsasının desteklediği tüm kripto varlık listesi yenilenmiş olur.



```

public async RefreshSymbols(request: Request, response: Response) {
    try {
        const { name } = request.body;

        const exchange = await Exchange.findOne({ name });

        if (!exchange) {
            return response.status(404).send(new FailureResult("Exchange not found."));
        }

        const exchangeInfoResponse = await axios(exchange.baseApi + exchange.symbolListEndpoint);

        let symbolArray: string[] = [];

        switch (name) {
            case CryptoExchange.Binance:
                for (const symbol of exchangeInfoResponse.data.symbols) {
                    if (symbol.symbol.endsWith("USDT")) {
                        symbolArray.push(symbol.symbol);
                    }
                }
                break;
            case CryptoExchange.KuCoin:
                for (const symbolObject of exchangeInfoResponse.data.data) {
                    if (symbolObject.symbol.endsWith("USDT")) {
                        symbolArray.push(symbolObject.symbol);
                    }
                }
                break;
            default:
                break;
        }

        exchange.symbols = symbolArray;
        await exchange.save();

        return response.status(200).send(new SuccessResult("Exchange Refreshed Successfully!",
            exchange));
    } catch (error: any) {
        console.log(error);
        return response.status(500).send(new FailureResult("Something went wrong."));
    }
}

```

Şekil 3.2.13. Refresh endpointinin kod tanımı

List

List işleminde veri tabanında kayıtlı olan tüm kripto varlıklar alınarak 200 status code'u ile birlikte kullanıcıya dönülür.



```

public async List(request: Request, response: Response) {
  try {
    const exchangeList = await Exchange.find();

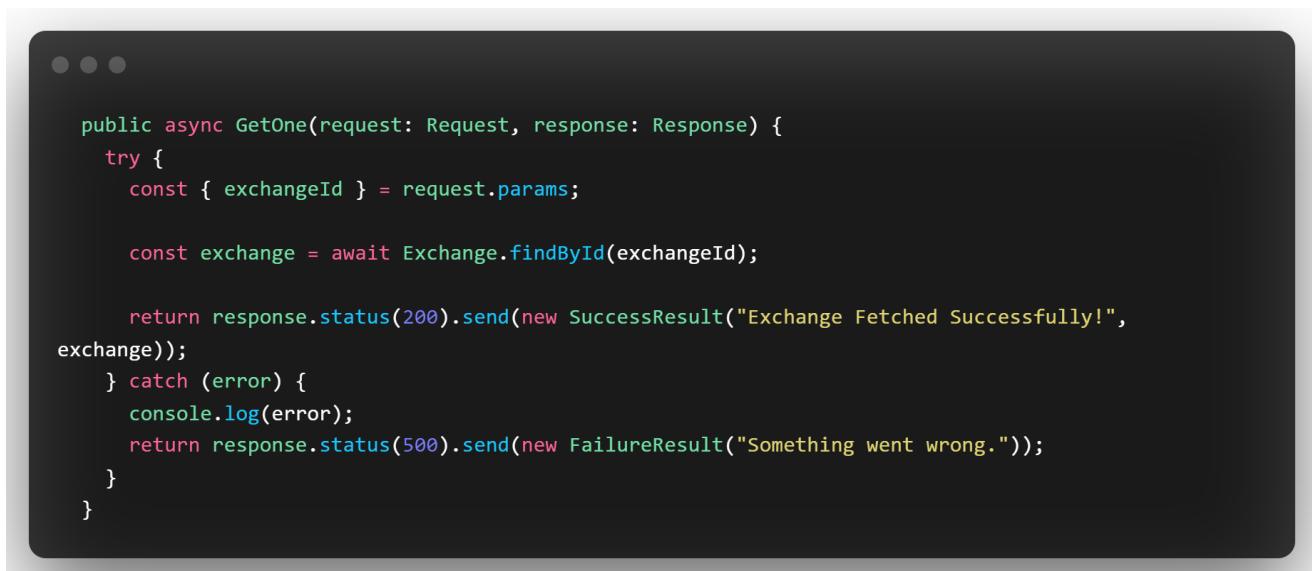
    return response.status(200).send(new SuccessResult("Exchange List Fetched Successfully!",
exchangeList));
  } catch (error) {
    console.log(error);
    return response.status(500).send(new FailureResult("Something went wrong."));
  }
}

```

Şekil 3.2.14. List endpointinin kod tanımı

Get One

Get One işleminde öncelikle parametre olarak verilen id bilgisi alınır. Alınan bu id bilgisi ile veri tabanında bir sorgu yapılır ve bu sorgu sonucu elimize gelen kripto borsası 200 status code'u ile birlikte kullanıcıya dönülür.



```

public async GetOne(request: Request, response: Response) {
  try {
    const { exchangeId } = request.params;

    const exchange = await Exchange.findById(exchangeId);

    return response.status(200).send(new SuccessResult("Exchange Fetched Successfully!",
exchange));
  } catch (error) {
    console.log(error);
    return response.status(500).send(new FailureResult("Something went wrong."));
  }
}

```

Şekil 3.2.15. GetOne endpointinin kod tanımı

UserCrypto Endpointleri

UserCrypto için kullanıcının cüzdanına yeni bir kripto varlık eklemesi için kullanılacak olan add endpointi, parametre olarak verilen id bilgisine göre bir kripto varlık üzerinde güncelleme yapılmasını sağlayacak olan update endpointi ve parametre olarak verilen id bilgisine göre bir kripto varlığı kullanıcının cüzdanından silmesi için kullanılacak olan delete endpointi olmak üzere 3 adet endpoint tanımlanmıştır.

```
import express from "express";
import { UserCryptoController } from "../../controllers/userCrypto";
import { Auth } from "../../middlewares/auth";

const router = express.Router();

router.post("/", Auth.authenticate, UserCryptoController.Add);
router.patch("/:userCryptoId", Auth.authenticate, UserCryptoController.Update);
router.delete("/:userCryptoId", Auth.authenticate, UserCryptoController.Delete);

export default router;
```

Şekil 3.2.16. UserCrypto endpointlerinin belirtildiği kod tanımı

Add

Create işleminde öncelikle *exchangeName*, *symbol*, *amount* verileri body'den alınır. Alınan bu veriler öncelikle belirtilen kurallar dahilinde validasyona sokulur. Eğer uygun veriler iletilmiş ise authenticate olmuş olan kullanıcının cüzdanı ve body içerisindeki *exchangeName* parametresi ile iletilmiş olan kripto borsası veri tabanından getirilir. Sonrasında kullanıcının iletilmiş olduğu sembolün ilgili kripto borsasında bulunup bulunmadığı kontrol edilir. Eğer bulunmuyorsa kullanıcıya ilgili mesaj gönderilir. Bulunuyorsa öncelikle ilgili kripto varlığın güncel değeri alınır.

```
public static async Add(request: Request, response: Response) {
  try {
    const { exchangeName, symbol, amount } = request.body;

    await userCryptoCreateValidator.validateAsync({ exchangeName, symbol, amount });

    const userWallet = await Wallet.findById(response.locals.user.walletId);

    if (!userWallet) {
      return response.status(404).send(new FailureResult("User Wallet not found"));
    }

    const exchange = await Exchange.findOne({ name: exchangeName });

    if (!exchange) {
      return response.status(404).send(new FailureResult("Exchange not found"));
    }

    if (!exchange.symbols.includes(symbol)) {
      return response.status(400).send(new FailureResult(symbol + " does not exist on " +
exchange.name));
    }

    const avgPrice = await CryptoPriceHelper.getPrice(
      exchange.baseApi + exchange.priceEndpoint + symbol,
      exchange.name
    );
  .
  .
}
```

Şekil 3.2.17. Add endpointinin kod tanımı - 1

Güncel değeri alındıktan sonra yeni bir UserCrypto kaydı oluşturulur. Yeni kaydın oluşturulması sonrası kullanıcının cüzdanı değeri güncellenir. Yeni kripto varlığın cüzdana eklenmesi sonucunda cüzdanı düzenli aralıklarla takip edecek olan RecurringJob kaydının enabled alanı true olacak şekilde güncellenir. 200 status code'u ile birlikte kullanıcıya eklenen kripto varlık ve cüzdanın güncel değeri döndürülür.

```

public static async Add(request: Request, response: Response) {
  .

  const newUserCrypto = await UserCrypto.create({
    walletId: userWallet.id,
    exchangeId: exchange._id,
    symbol,
    firstPrice: avgPrice,
    lastPrice: avgPrice,
    amount,
  });

  userWallet.cryptoIds.push(newUserCrypto.id);
  userWallet.balance += +(avgPrice * amount).toFixed(4);
  userWallet.balance = +userWallet.balance.toFixed(4);
  await userWallet.save();

  if (userWallet.cryptoIds.length >= 1) {
    await RecurringJobModel.findByIdAndUpdate(userWallet.recurringJobId, { $set: { enabled: true } });
  }
}

newUserCrypto.exchange = exchange;
return response
  .status(200)
  .send(
    new SuccessResult("Crypto added successfully", { newUserCrypto: newUserCrypto, balance: userWallet.balance })
  );
} catch (error: any) {
  if (error.isJoi) {
    return response.status(400).send(new FailureResult("Validation error: " + error.message));
  }

  console.log(error);
  return response.status(500).send(new FailureResult("Something went wrong."));
}
}

```

Şekil 3.2.18. Add endpointinin kod tanımı - 2

Update

Update işleminde öncelikle *newAmount* ve *userCryptoId* verileri body'den ve parametre olarak alınır. Alınan *userCryptoId* ile ilgili UserCrypto kaydı ve bağlı olduğu cüzdan kaydı veri tabanından alınır. Sonrasında ilgili kripto varlığın yeni miktarına göre güncel değeri hesaplanır ve ilgili kayıt buna göre güncellenir. Yeni değer ile kullanıcının cüzdanı da güncellendikten sonra 200 status code'u ile birlikte kullanıcıya güncel kripto varlık kaydı dönülür.



```

public static async Update(request: Request, response: Response) {
    try {
        const { newAmount } = request.body;
        const { userCryptoId } = request.params;

        const userCrypto = await UserCrypto.findById(userCryptoId).populate("exchange", "name baseApi
priceEndpoint");

        if (!userCrypto) {
            return response.status(404).send(new FailureResult("Specified user crypto not found!"));
        }

        const wallet = await Wallet.findById(userCrypto.walletId);

        if (!wallet) {
            return response.status(404).send(new FailureResult("Wallet not found!"));
        }

        if (response.locals.user.walletId.toString() !== wallet._id.toString()) {
            return response.status(403).send(new FailureResult("You cannot do that!"));
        }

        wallet.balance -= +(userCrypto.lastPrice * userCrypto.amount).toFixed(4);
        wallet.balance = +wallet.balance.toFixed(4);
        const avgPrice = await CryptoPriceHelper.getPrice(
            userCrypto.exchange.baseApi + userCrypto.exchange.priceEndpoint + userCrypto.symbol,
            userCrypto.exchange.name
        );

        userCrypto.lastPrice = avgPrice;
        wallet.balance += +(avgPrice * newAmount).toFixed(4);
        wallet.balance = +wallet.balance.toFixed(4);
        await wallet.save();

        userCrypto.amount = newAmount;
        await userCrypto.save();

        return response.status(200).send(new SuccessResult("Crypto Updated Successfully!",
userCrypto));
    } catch (error) {
        console.log(error);
        return response.status(500).json(new FailureResult("Something went wrong."));
    }
}

```

Şekil 3.2.19. Update endpointinin kod tanımı

Delete

Update işleminde öncelikle *userCryptoId* verisi body'den alınır. Alınan *userCryptoId* ile ilgili UserCrypto kaydı ve bağlı olduğu cüzdan kaydı veri tabanından alınır. Sonrasında silinecek olan kripto varlığın son fiyatı ile miktarı çarpılarak oluşan değer ilgili cüzdanın düşülür. Silme işlemi sonrası cüzdana başka bir kripto varlık kalmadı ise cüzdanı düzenli aralıklarla takip edecek olan RecurringJob kaydının enabled alanı false olacak şekilde güncellenir. Böylece içinde hiç kripto varlık olmayan bir cüzdan Job Handler tarafından işlenmemiş olur.

```
...
public static async Delete(request: Request, response: Response) {
  try {
    const { userCryptoId } = request.params;

    const userCrypto = await UserCrypto.findById(userCryptoId);

    if (!userCrypto) {
      return response.status(404).send(new FailureResult("Specified user crypto not found!"));
    }

    const wallet = await Wallet.findById(userCrypto.walletId);

    if (!wallet) {
      return response.status(404).send(new FailureResult("Wallet not found!"));
    }

    if (response.locals.user.walletId.toString() !== wallet._id.toString()) {
      return response.status(403).send(new FailureResult("You cannot do that!"));
    }

    wallet.balance -= +(userCrypto.lastPrice * userCrypto.amount).toFixed(4);
    wallet.balance = +wallet.balance.toFixed(4);
    wallet.cryptoIds = wallet.cryptoIds.filter((cryptoId) => cryptoId.toString() !== userCrypto.id);

    await wallet.save();

    if (wallet.cryptoIds.length === 0) {
      await RecurringJobModel.findByIdAndUpdate(wallet.recurring jobId, { $set: { enabled: false } });
    }
  }

  await userCrypto.deleteOne();

  return response.status(204).send();
} catch (error) {
  console.log(error);
  return response.status(500).json(new FailureResult("Something went wrong."));
}
}
```

Sekil 3.2.20. Delete endpointinin kod tanımı

BÖLÜM 4. JOB HANDLER SERVİSİ

Bu servis; TypeScript, MongoDB, Cron Expression ve NodeJS kullanılarak geliştirilmiştir. Servisin amacı; belirli zaman aralıklarıyla çalışıp, kullanıcıların cüzdanlarında bulunan kripto varlıkların anlık değerlerini kontrol ederek, kullanıcılara toplam cüzdan değerlerinin zamana bağlı değişimini takip edebilmelerine olanak sağlamaktır.

4.1. Veritabanı Modelleri

Job Handler servisi düzenli aralıklarla çalıştığında hangi Job'ların çalışacağını belirlemek adıma aşağıdaki model tasarlanmıştır. Her Wallet oluşturulduğunda o Wallet'in referans verdiği bir de RecurringJob oluşturulur. Böylece her RecurringJob sadece 1 Wallet ile ilişkilendirilmiş olur.

- RecurringJob Modeli
 - schedule // cron expression
 - lastRunAt
 - nextRunAt
 - beingTriggered
 - beingTriggeredAt
 - enabled
- ChangeRecord Modeli
 - eventDate
 - value
 - walletId

İşleyiş

1- Uygulama her 10 saniyede bir çalışıp, veri tabanındaki RecurringJob modelinde bulunan kayıtlardan nextRunAt alanındaki tarih verisi, uygulamanın çalışma tarihinden küçük veya eşit olan kayıtları seçer.



```

async function getJobsToRun() {
  const now = new Date();
  const twoMinutesAgo = new Date(now.getTime() - 2 * 60 * 1000);

  await RecurringJobModel.updateMany(
    {
      nextRunAt: { $lte: new Date() },
      enabled: true,
      $or: [{ beingTriggered: false }, { beingTriggered: true, beingTriggeredAt: { $lt: twoMinutesAgo } }]
    },
    {
      $set: { beingTriggered: true, beingTriggeredAt: now, lastRunAt: now },
    }
  );

  const jobsToRun = await RecurringJobModel.find({ beingTriggeredAt: now });
  return jobsToRun;
}

```

Şekil 4.1.1. getJobsToRun fonksiyonu tanımı

2- Seçilen tüm Job'ların paralel olarak çalışabilmesi için tüm Job'lar maplenerek bir Promise dizisi içine alınır ve sonrasında bu dizi await edilir.



```

async function checkAndRunRecurringJobs() {
  const jobsToRun = await getJobsToRun();

  console.log("running " + jobsToRun.length + " jobs");

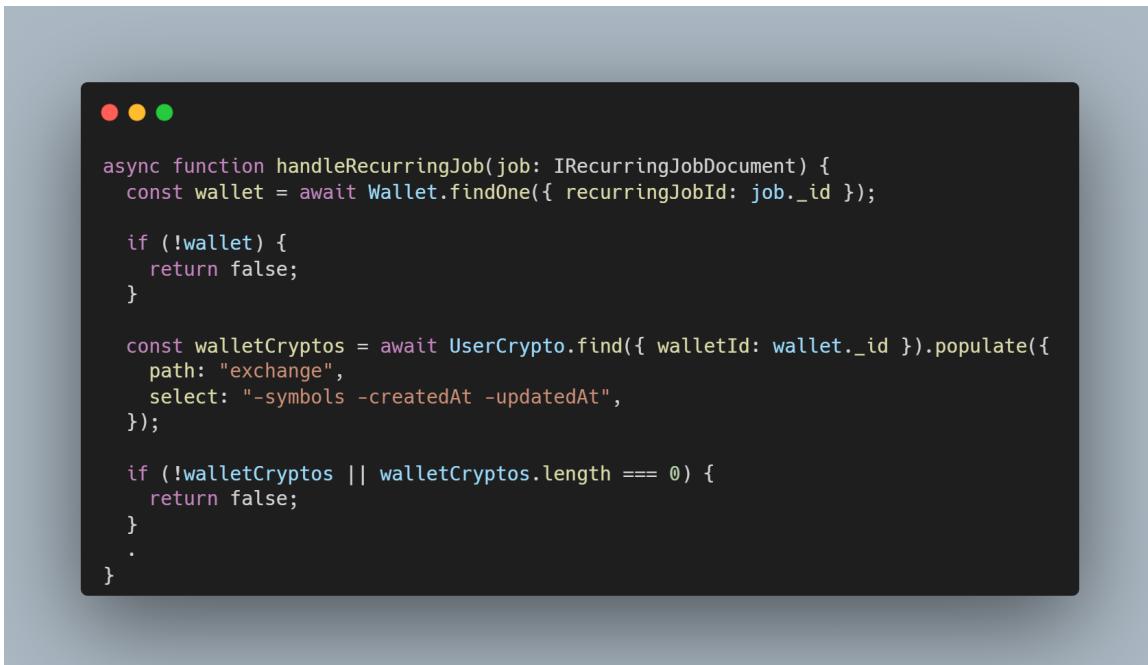
  const jobPromises = jobsToRun.map((job) => {
    return handleRecurringJob(job);
  });

  await Promise.all(jobPromises);
}

```

Şekil 4.1.2. Job'ların paralel olarak çalıştırılmasını sağlayan kod tanımı

3- Her bir Job'ın tek tek işlendiği handleRecurringJob fonksiyonunda öncelikle Job'a bağlı olan Wallet ve sonrasında da o Wallet'a dahil olan UserCrypto'ları veri tabanından çekip, var olup olmadıkları kontrol edilir.



```

async function handleRecurringJob(job: IRecurringJobDocument) {
  const wallet = await Wallet.findOne({ recurring jobId: job._id });

  if (!wallet) {
    return false;
  }

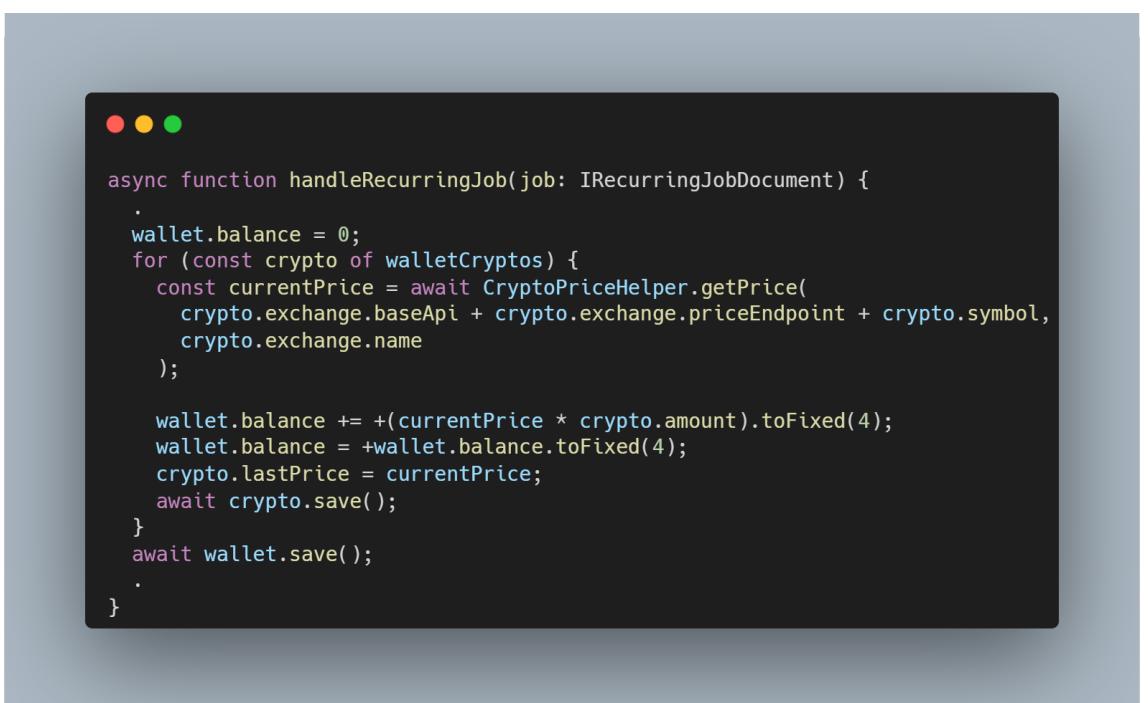
  const walletCryptos = await UserCrypto.find({ walletId: wallet._id }).populate({
    path: "exchange",
    select: "-symbols -createdAt -updatedAt",
  });

  if (!walletCryptos || walletCryptos.length === 0) {
    return false;
  }
}

```

Şekil 4.1.3. wallet ve walletCryptos kontrollerinin yapıldığı kod tanımı

4- Bu verilerin varlığını doğruladıktan sonra her bir walletCrpts için tek tek anlık price değer alınır ve ilgili alanlar güncel veriye uygun olacak şekilde güncellenir.



```

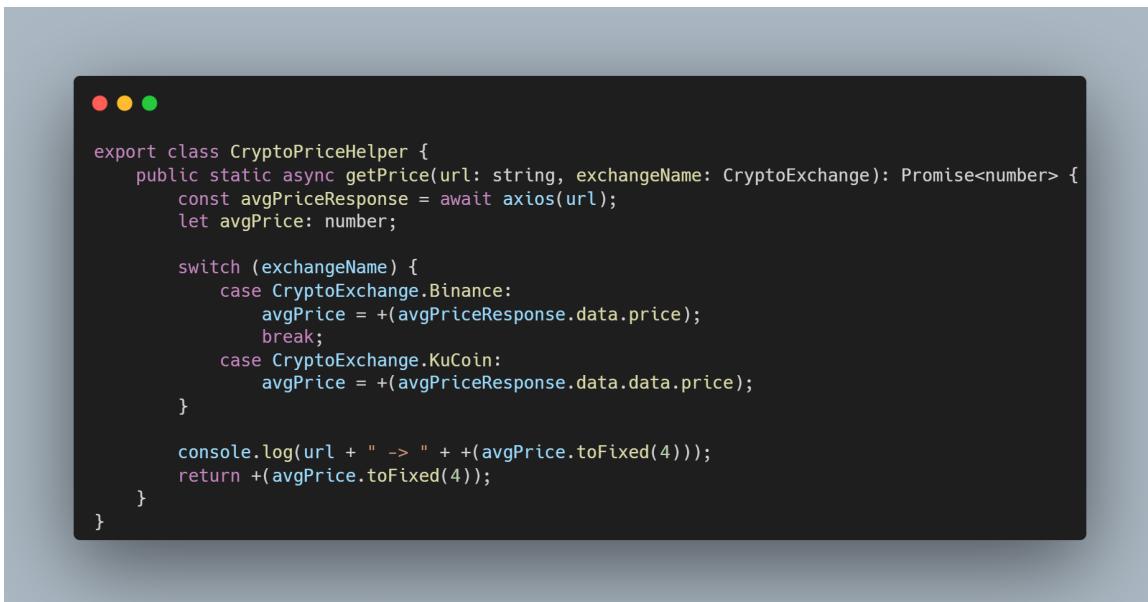
async function handleRecurringJob(job: IRecurringJobDocument) {
  .
  .
  wallet.balance = 0;
  for (const crypto of walletCryptos) {
    const currentPrice = await CryptoPriceHelper.getPrice(
      crypto.exchange.baseApi + crypto.exchange.priceEndpoint + crypto.symbol,
      crypto.exchange.name
    );

    wallet.balance += +(currentPrice * crypto.amount).toFixed(4);
    wallet.balance = +wallet.balance.toFixed(4);
    crypto.lastPrice = currentPrice;
    await crypto.save();
  }
  await wallet.save();
}

```

Şekil 4.1.4. Her crypto için price değerinin güncellendiği kod tanımı

5- Güncel price değerinin alınması için CryptoPriceHelper sınıfının kullanılan getPrice adındaki statik metodu kullanılır.



```

export class CryptoPriceHelper {
    public static async getPrice(url: string, exchangeName: CryptoExchange): Promise<number> {
        const avgPriceResponse = await axios(url);
        let avgPrice: number;

        switch (exchangeName) {
            case CryptoExchange.Binance:
                avgPrice = +(avgPriceResponse.data.price);
                break;
            case CryptoExchange.KuCoin:
                avgPrice = +(avgPriceResponse.data.data.price);
        }

        console.log(url + " -> " + +(avgPrice.toFixed(4)));
        return +(avgPrice.toFixed(4));
    }
}

```

Şekil 4.1.5. Her crypto için price değerinin güncellendiği kod tanımı

6- Güncel price değeri kaydedildikten sonra Job'ın nextRunAt ve beingTriggered alanları güncellenir. Böylece ilgili Job'ın işinin tamamlandığı ve sıradaki çalışma için nextRunAt alanındaki tarihin gelmesi beklentiği belirtilmiştir olur.



```

async function handleRecurringJob(job: IRecurringJobDocument) {
    .

    .

    const parsed = parseExpression(job.schedule);
    const nextRunAt = parsed.next().toDate();

    job.beingTriggered = false;
    job.nextRunAt = nextRunAt;
    await job.save();

    return true;
}

```

Şekil 4.1.6. Job'ın alanlarının güncellendiği kod tanımı

BÖLÜM 5. FRONTEND (REACT.JS)

Frontend (Ön yüz) geliştirmesi React, TypeScript, TailwindCss, Mantine.dev, Toaster kütüphaneleri yardımıyla yapılmıştır.

Register (Kayıt Ol) Sayfası

Bu sayfada gösterilen form vesilesi ile kullanıcıdan alınan E-mail ve Parola bilgileri ile -eğer o maile kayıtlı başka bir hesap yoksa- kullanıcı oluşturulur.

Bu sayfanın tasarımları için Tailwind Css ve Mantine kullanılmıştır. Mantine'ın useForm hook'u form yapısının kurulmasını çok kolaylaştırmıştır.

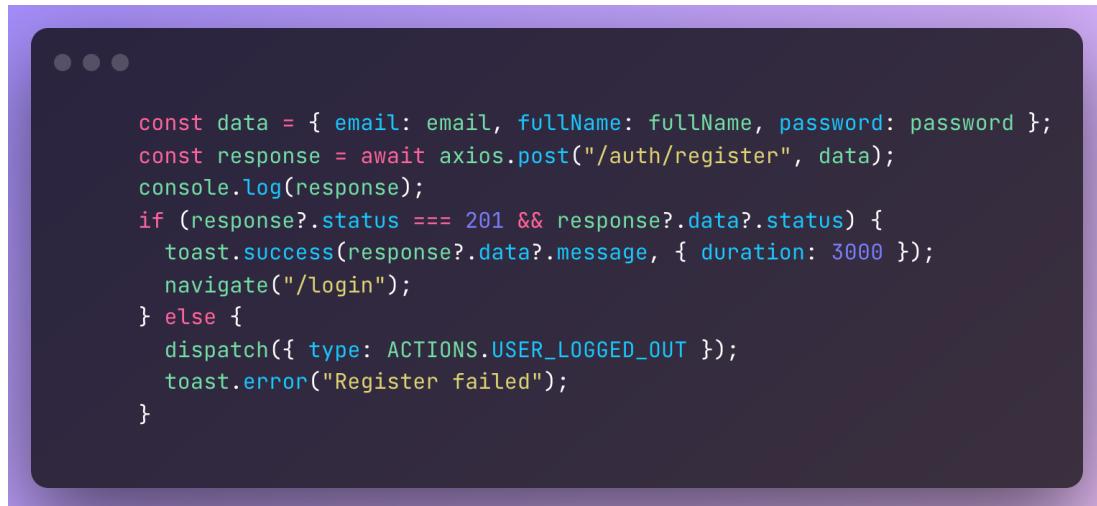


```
const form = useForm({
  initialValues: { email: "", fullName: "", password: "secret", confirmPassword: "sevret" },
  validate: {
    email: (value) => (/^\S+@\S+$/.test(value) ? null : "Invalid email"),
    fullName: (value) => (value.length > 0 ? null : "Full name is required"),
    password: (value: string) => (value.length > 4 ? null : "Password must be at least 4 characters"),
    confirmPassword: (value, values) => (value !== values.password ? "Passwords did not match" : null),
  },
});
```

Şekil 5.1.1. Mantine'ın Form yapısı ve Register için oluşturulan yapı

Register sayfasında kullanıcının kayıt olması için E-Mail, Full Name (isim soyisim), Password (şifre) bilgileri alınmaktadır. Bir de şifre doğrulaması için confirm password ile şifre tekrar istenmektedir.

Kurallara uygun olarak alınan bilgiler axios kütüphanesi ile server'a gönderilir. Serverdan gelen cevaba göre aksiyon alınır.



Şekil 5.1.2. Register için server'a gönderilen post yapısı ve sonrası

Eğer gelen cevap olumluysa login sayfasına yönlendirme yapılır ve sayfanın sağ üst kısmında Kullanıcı Başarı ile oluşturuldu uyarısı dönülür.

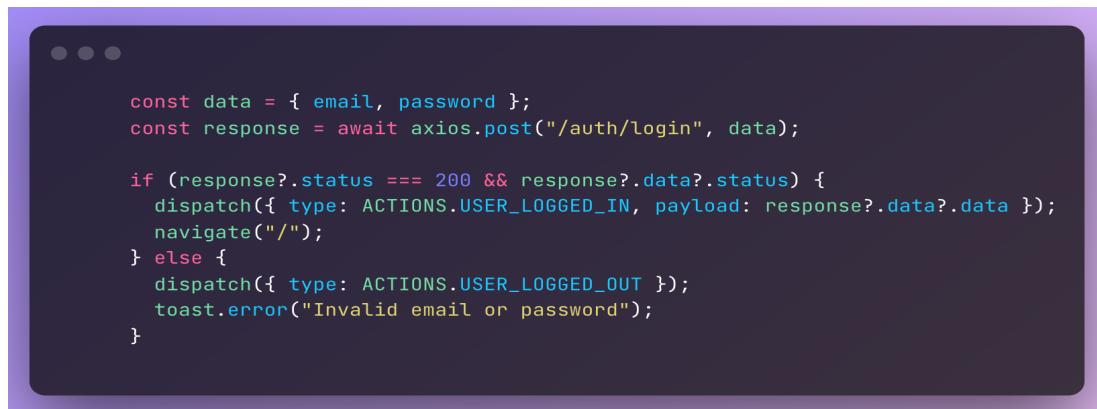
Login (Giriş Yap) Sayfası

Bu sayfada gösterilen form vesilesi ile kullanıcıdan alınan E-mail ve Parola bilgileri server tarafına gönderilir. Eğer veritabanında kayıtlı bir e-mail varsa ve bu e-mailin parolası ile kullanıcının girdiği parola eşlenir ise kullanıcı direkt olarak Dashboard (Gösterge Paneli) sayfasına yönlendirilir. Eğer bilgilerde yanlışlık var ise Toaster kütüphanesi yardımı ile oluşturulan hata mesajları sayfanın sağ üst tarafında birkaç saniye görülür.



```
const form = useForm({
  initialValues: { email: "", password: "secret" },
  validate: {
    email: (value) => (/^[\S+@\S+$/.test(value) ? null : "Invalid email"),
    password: (value: string) => (value.length > 4 ? null : "Password must be at least 6
characters"),
  },
});
```

Şekil 5.2.1. Login sayfası form yapısı



```
const data = { email, password };
const response = await axios.post("/auth/login", data);

if (response?.status === 200 && response?.data?.status) {
  dispatch({ type: ACTIONS.USER_LOGGED_IN, payload: response?.data?.data });
  navigate("/");
} else {
  dispatch({ type: ACTIONS.USER_LOGGED_OUT });
  toast.error("Invalid email or password");
}
```

Şekil 5.2.2. Login sayfası için ana login fonksiyonu



```
const response = await axios.get("/auth/me");
if (response?.status === 200 && response?.data?.status) {
  dispatch({ type: ACTIONS.USER_LOGGED_IN, payload: response?.data?.data });
}
```

Şekil 5.2.3. Login sayfası giriş yapılmış yapılmadığının kontrolünü yapan fonksiyon

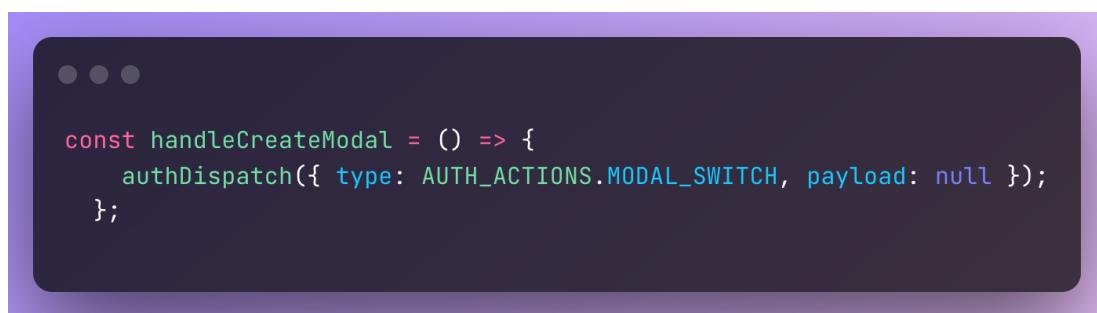
Navbar Yapısı

Navbar, kullanıcı başarı ile giriş yaptıktan sonra public olmayan protected sayfaların hepsinde sayfanın üst tarafında görülmektedir.

En sol tarafta uygulamanın simgesi, ortanın biraz solunda menüler ve en sağda kullanıcı bilgileri ile Logout butonu vardır.

Crypto ekleme butonu da navbara eklenmiştir. Bu butona basınca authDispatch fonksiyonu çağırılır ve içerisinde modal_switch emiri verilir. Bu emir false olan modal_switch state’ini tersine çevirir ve bu sayede main sayfasında kripto varlık ekleme için bilgileri alacak olan modal sayfası açılır.

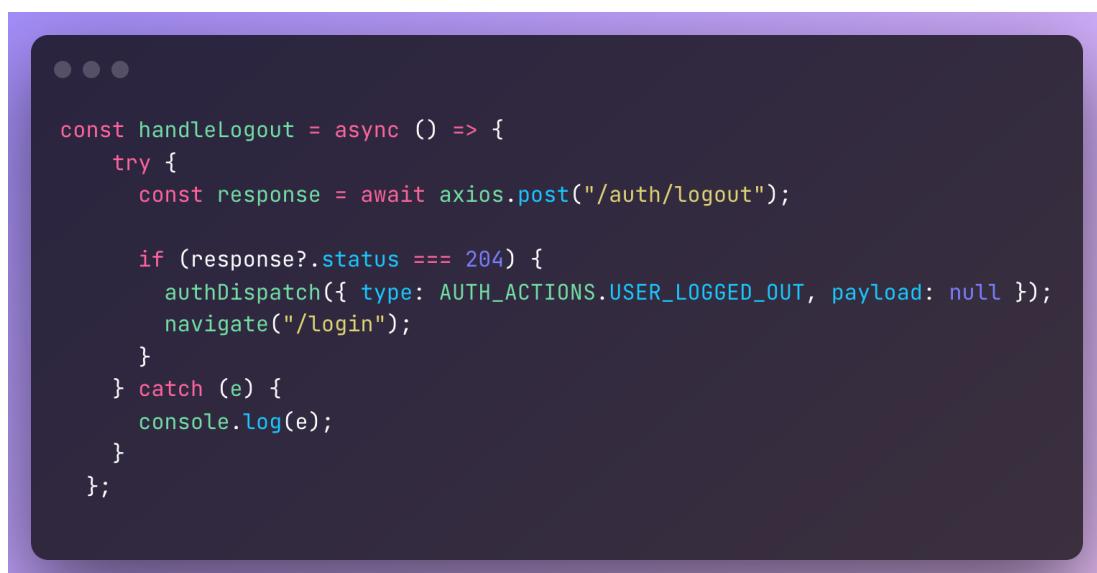
Navbar tasarımlı Tailwind Css ve mantine ile yapılmıştır.



```
const handleCreateModal = () => {
  authDispatch({ type: AUTH_ACTIONS.MODAL_SWITCH, payload: null });
};
```

Şekil 5.3.1. Anasayfada modal açılması için emir giren fonksiyon

Navbar’ın en sağ tarafında Logout butonu vardır. Bu buton ile kişi hesabından çıkış yapar ve login sayfasına yönlendirilir.



```
const handleLogout = async () => {
  try {
    const response = await axios.post("/auth/logout");

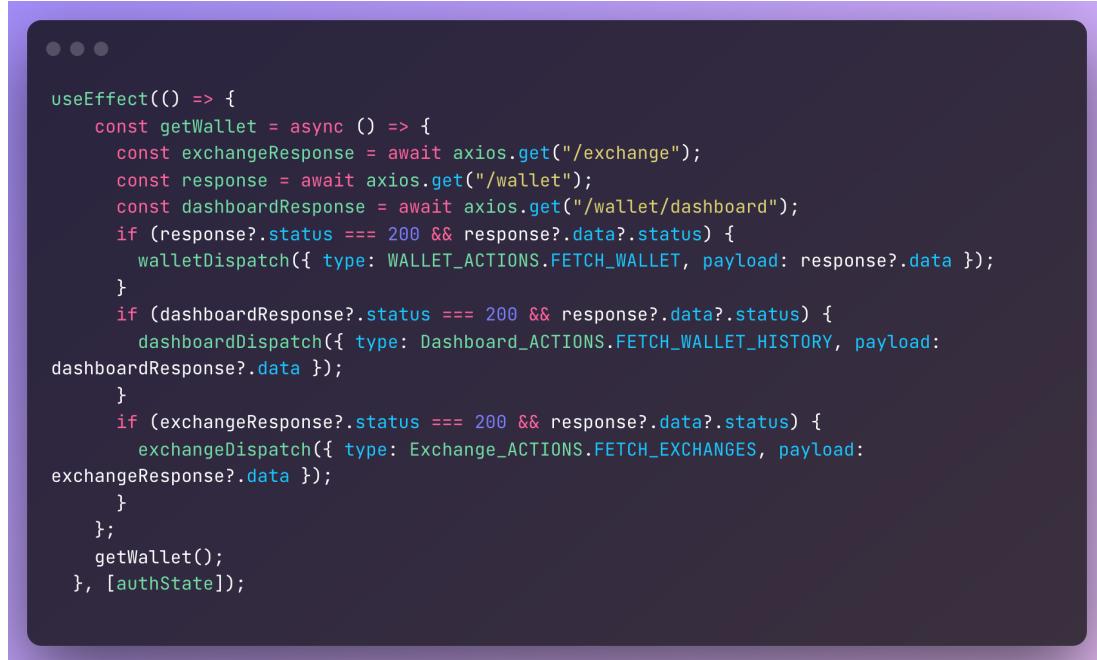
    if (response?.status === 204) {
      authDispatch({ type: AUTH_ACTIONS.USER_LOGGED_OUT, payload: null });
      navigate("/login");
    }
  } catch (e) {
    console.log(e);
  }
};
```

Şekil 5.3.2. Anasayfada modal açılması için emir giren fonksiyon

Main Sayfası

Login yapıldıktan sonra yönlendirilen sayfadır. Tüm bilgiler burada yer alır. Bu sayfada kişinin toplam bütçesi, elindeki varlıklar ve toplam bütçesinin zaman içerisindeki değişimi yer alır.

Main sayfasına girildiği anda bu bilgiler serverdan çekilir ve Context'te tutulur.



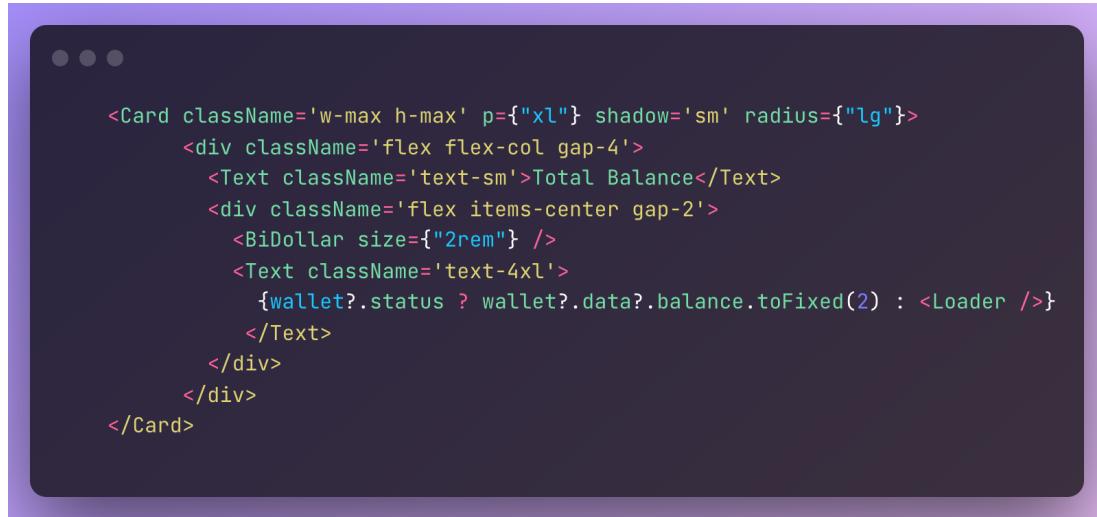
```
useEffect(() => {
  const getWallet = async () => {
    const exchangeResponse = await axios.get("/exchange");
    const response = await axios.get("/wallet");
    const dashboardResponse = await axios.get("/wallet/dashboard");
    if (response?.status === 200 && response?.data?.status) {
      walletDispatch({ type: WALLET_ACTIONS.FETCH_WALLET, payload: response?.data });
    }
    if (dashboardResponse?.status === 200 && response?.data?.status) {
      dashboardDispatch({ type: Dashboard_ACTIONS.FETCH_WALLET_HISTORY, payload: dashboardResponse?.data });
    }
    if (exchangeResponse?.status === 200 && response?.data?.status) {
      exchangeDispatch({ type: Exchange_ACTIONS.FETCH_EXCHANGES, payload: exchangeResponse?.data });
    }
  };
  getWallet();
}, [authState]);
```

Şekil 5.4.1. Main sayfasına giriş yapıldığında serverdan çekilen dataların Context'e aktarılması

Balance Component

Main sayfasının en üst sağ kısmında Toplam Varlık yazmaktadır. Bu kısım Mantine kütüphanesi içinde yer alan Card Component'i içinde yapılmıştır. İçindeki Dolar işaretini react-icons kütüphanesinden alınmıştır.

Data, Wallet Context Provider'ından çekilmiştir. Balance verisi çekildikten sonra .toFixed(2) yaparak gelen değerin noktadan sonra 2 basamağının alınması sağlanmıştır.



```
<Card className='w-max h-max p={"xl"} shadow='sm' radius='lg'>
  <div className='flex flex-col gap-4'>
    <Text className='text-sm'>Total Balance</Text>
    <div className='flex items-center gap-2'>
      <BiDollar size={2rem} />
      <Text className='text-4xl'>
        {wallet?.status ? wallet?.data?.balance.toFixed(2) : <Loader />}
      </Text>
    </div>
  </div>
</Card>
```

Şekil 5.4.2. Toplam değerinin yazarı olduğu Card componentinin yazılımı

User's Assets Component

Main sayfasındaki bir diğer component kullanıcının tüm kripto varlıklarının bulunduğu User's Assets bölümüdür. Bu bölüm için dışında bir Card yapısının yanı sıra içinde de her bir varlığın bilgileri alınıp SingleCrypto adında bir Component aktarılmıştır. Wallet Content Provider içinden dizi olarak alınan varlıkların her biri map fonksiyonu ile dönültür ve bilgileri SingleCrypto fonksiyonuna aktarılır.

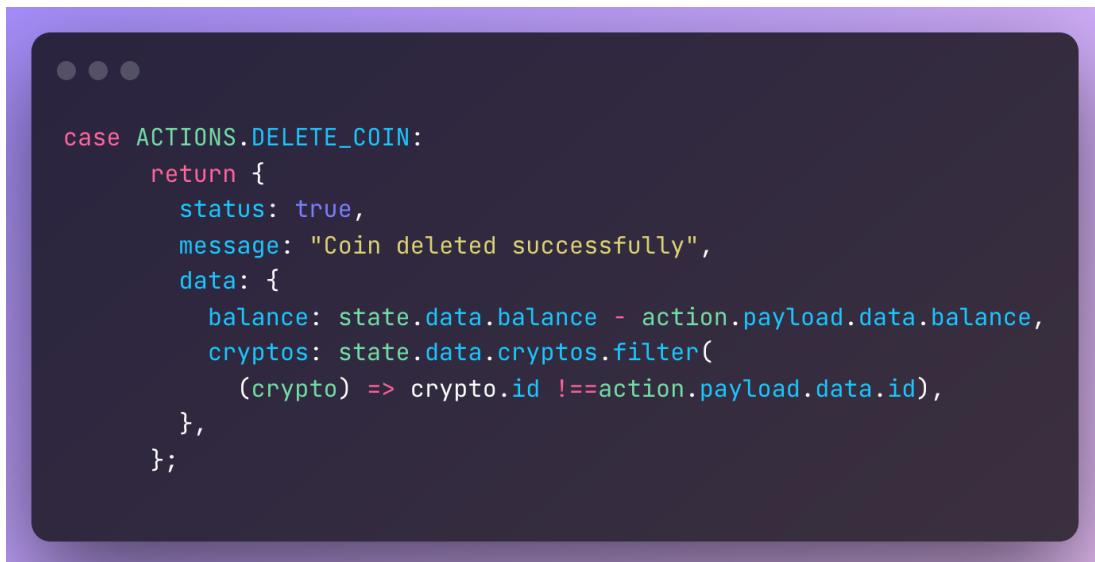


```
<Card className=' h-max flex flex-col items-center gap-5 w-full' p='xl' shadow='sm' radius='lg'>
  <Text size='xl'>User's Assets</Text>
  {wallet?.status ? (
    wallet?.data?.cryptos?.map?.((crypto) => (
      <SingleCrypto
        key={crypto?.id}
        id={crypto?.id}
        amount={crypto?.amount}
        exchangeLogo={crypto?.exchange?.logoUrl}
        exchangeName={crypto?.exchange?.name}
        lastPrice={crypto?.lastPrice}
        symbol={crypto?.symbol}
      />
    ))
  ) : (
    <Loader />
  )}
</Card>
```

Şekil 5.4.3. User's Assets Componenti

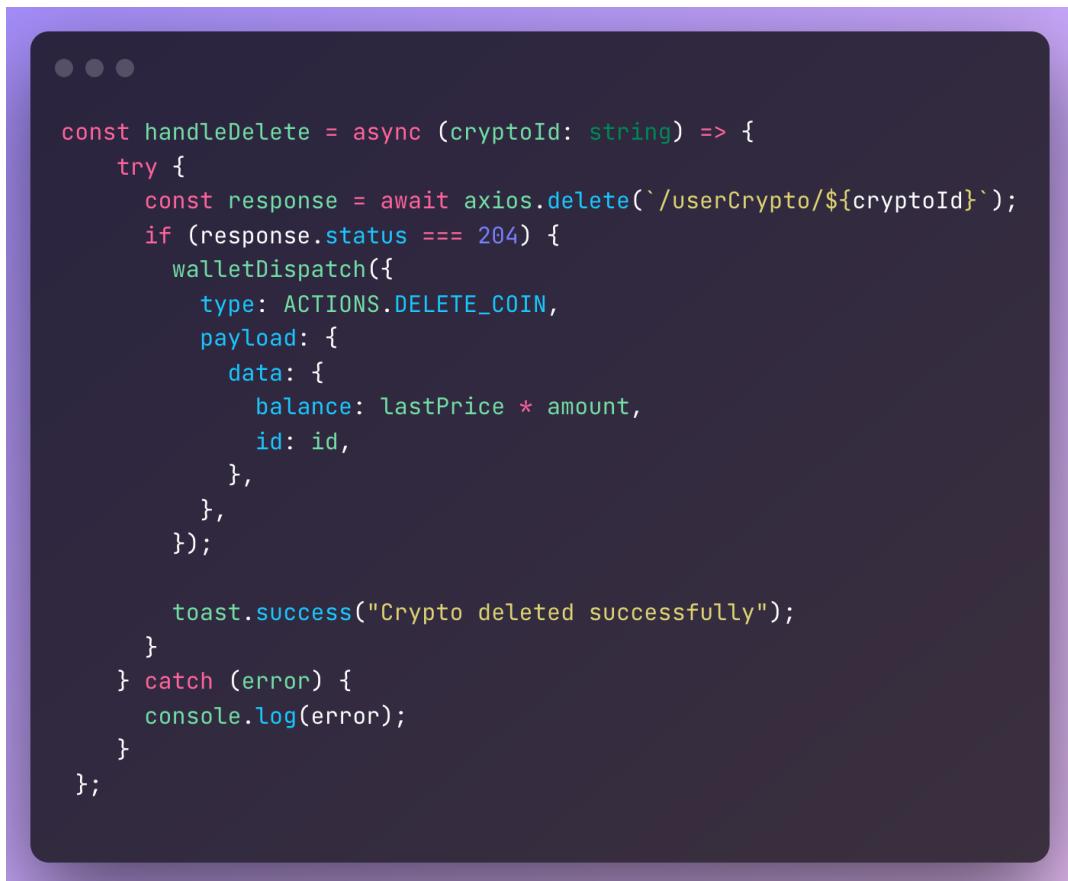
SingleCrypto Component

Bu component aldığı Logo, symbol, exchangeName, lastPrice, amount gibi bilgiler ile kripto varlığın gösterilmesini sağlar. Aynı zamanda en sağ tarafta 3 nokta iconu vardır. Bu ikona tıklayınca menü açılır. Bu menü içerisinde kırmızı bir Delete(sil) butonu ve sarı bir Edit(düzenle) butonu vardır. Delete butonu ile kripto varlık silinir.



Şekil 5.4.4. Kripto Varlığı Wallet Provider içinden silme

Bu işlem için önce server'a silme isteği atılır. Daha sonra kripto varlık Provider içerisinde silinir.



```

const handleDelete = async (cryptoId: string) => {
  try {
    const response = await axios.delete(`/userCrypto/${cryptoId}`);
    if (response.status === 204) {
      walletDispatch({
        type: ACTIONS.DELETE_COIN,
        payload: {
          data: {
            balance: lastPrice * amount,
            id: id,
          },
        },
      });
    }
    toast.success("Crypto deleted successfully");
  } catch (error) {
    console.log(error);
  }
};

```

Şekil 5.4.5. Kripto Varlığın server içerisinde silinmesi

Edit butonu ile component Edit moduna geçer ve arka planı sarı olur. Amount kısmı Mantine kütüphanesinden aldığım NumberInput componenti olur. Bu sayede kullanıcından yeni bir değer alınır ve Update butonuna basılarak (edit butonu yerine gelir) hem server hem de Provider tarafından varlığın güncellemesi gerçekleşir.



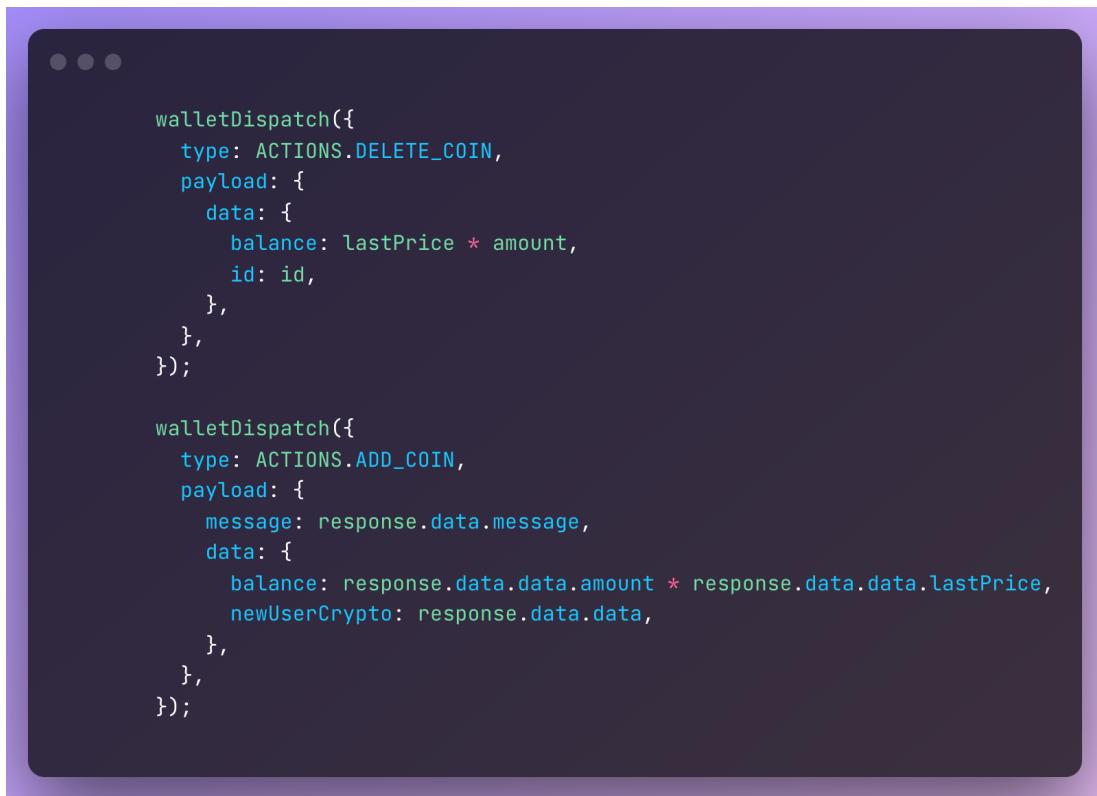
```

const response = await axios.patch(`/userCrypto/${cryptoId}`,
  { newAmount: newAmount });

```

Şekil 5.4.7. Kripto Varlığın server tarafında update'lenmesi

Provider tarafında kripto varlığın updatelenmesi için özel bir fonksiyon yoktur. Varlık, önce silinir daha sonra yeni değeri ile yeniden eklenir.



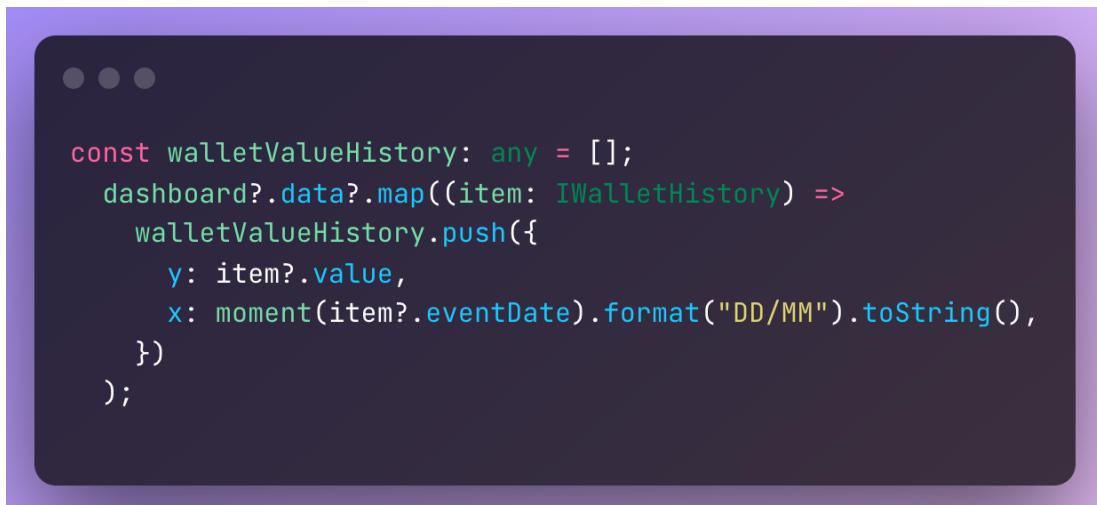
```
walletDispatch({
  type: ACTIONS.DELETE_COIN,
  payload: {
    data: {
      balance: lastPrice * amount,
      id: id,
    },
  },
});

walletDispatch({
  type: ACTIONS.ADD_COIN,
  payload: {
    message: response.data.message,
    data: {
      balance: response.data.data.amount * response.data.data.lastPrice,
      newUserCrypto: response.data.data,
    },
  },
});
```

Şekil 5.4.8. Kripto Varlığın Provider tarafında updatelenmesi (silinip eklenmesi)

Günlük Değişim Grafiği

Uygulamada grafik için nivo.rock kütüphanesinin ResponsiveLine Componenti kullanıldı. İçerisinde birkaç konfigürasyon yapıldı ve data eklendi. İstenilen tarih ve balance datası componentin isteğine göre manipüle edildi.



```
const walletValueHistory: any = [];
dashboard?.data?.map((item: IWalletHistory) =>
  walletValueHistory.push({
    y: item?.value,
    x: moment(item?.eventDate).format("DD/MM").toString(),
  })
);
```

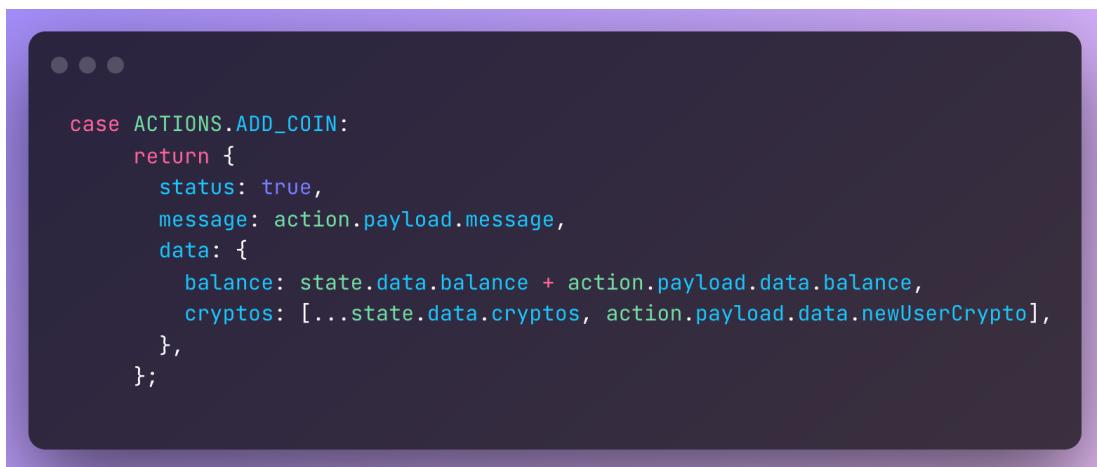
Şekil 5.4.9 Datanın grafik için istenilen hale getirilmesi

Yeni Kripto Ekleme Modali

Bu modal, daha önce Navbar kısmında bahsedildiği gibi Add Asset butonuna basıldığında açılır. Add Asset butonuna basıldığında auth Provider içerisindeki state true olur ve Modal açılır. Modal içerisinde önce Exchange seçilir. Exchange seçildikten sonra o Exchange içinde bulunan Kripto Varlıklar diğer select içerisinde belirir ve kullanıcı orada istediği varlığı seçer. Son olarak ne kadar varlık aldığıni giren kullanıcı Add butonuna basar ve varlık hem server hem Provider tarafında kullanıcının cüzdanına eklenir.



Şekil 5.4.10 Varlık eklenmesinin server tarafa istek olarak gönderilmesi ve Provider fonksiyonunun çağırılması

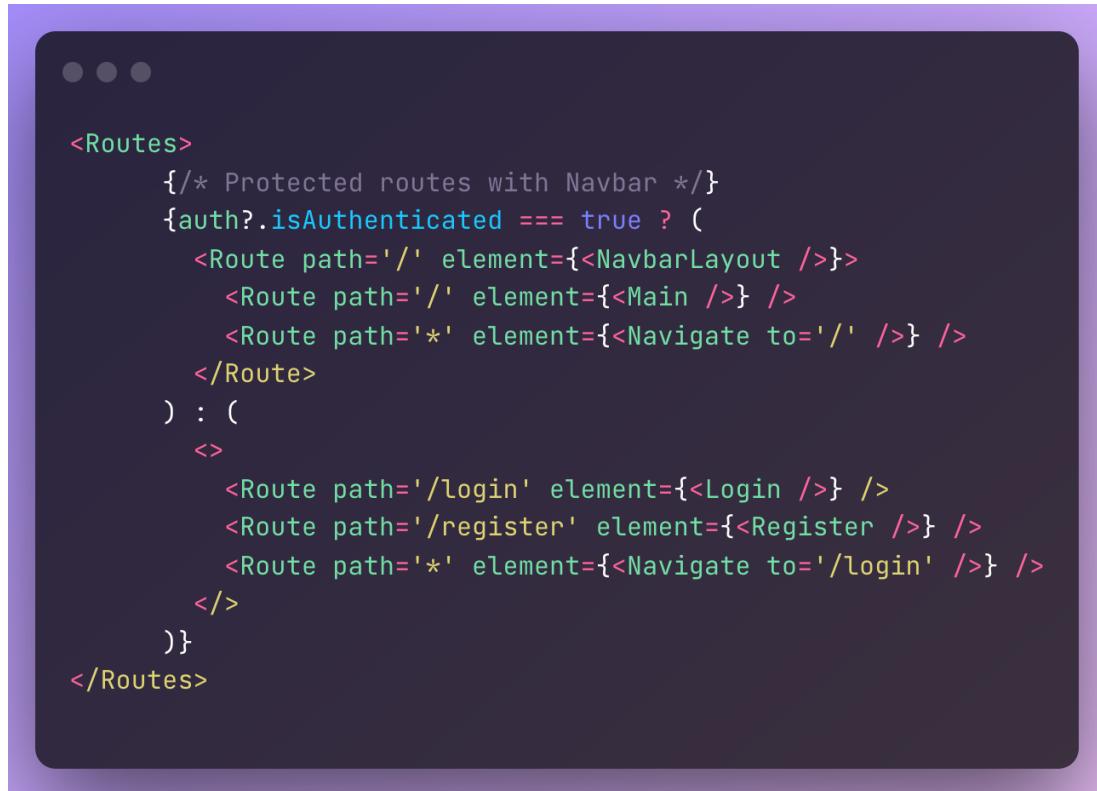


Şekil 5.4.11 Provider içerisindeki ADD_COIN fonksiyonu

Sayfa Yönlendirme

Uygulamada sayfa yönlendirmeleri App dosyası içerisinde react-router-dom kütüphanesi ile yapılır.

NavbarLayout Route içerisinde yazılan sayfalarda Navbar da olur.



The screenshot shows a code editor window with a dark theme. The code is written in JSX, defining a routes structure. It includes conditional logic based on the user's authentication status (isAuthenticated). If authenticated, it renders a NavbarLayout component as the parent route for paths '/' and '/'. For other paths, it uses a Navigate component to redirect to the '/login' page. If the user is not authenticated, it renders three separate routes for '/login', '/register', and other paths that also redirect to '/login'.

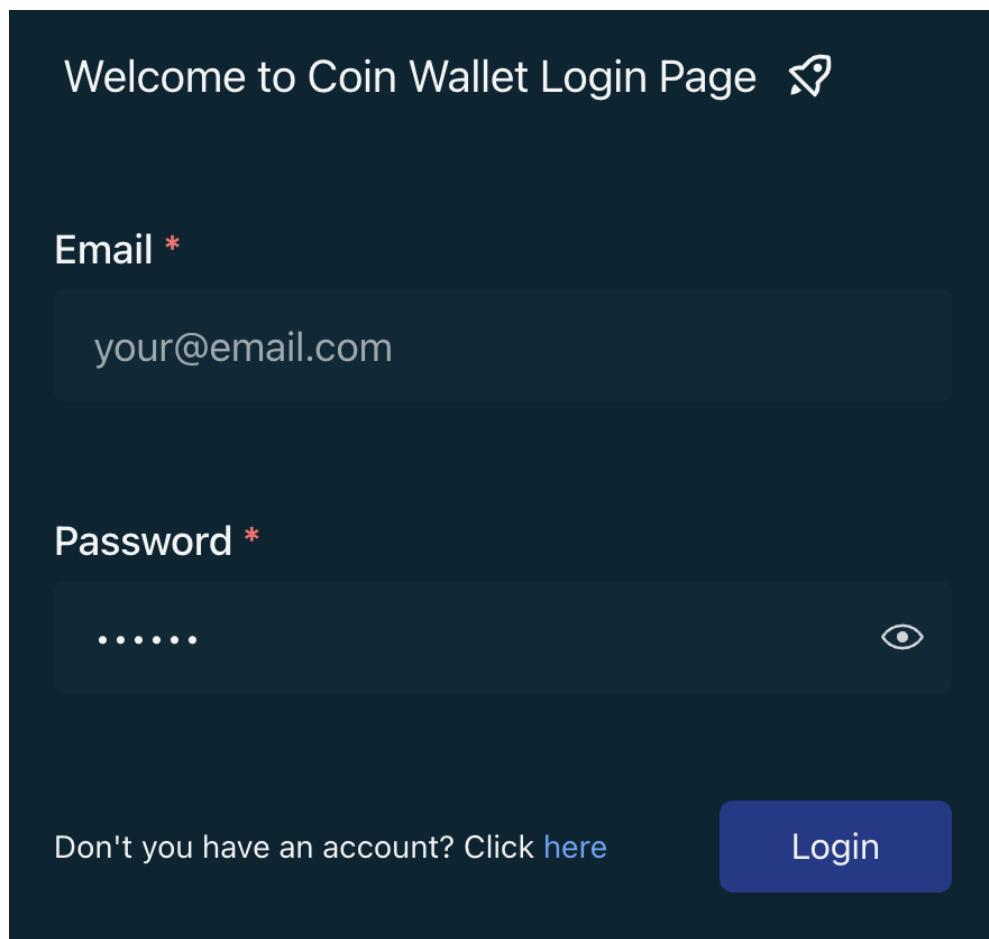
```
<Routes>
  {/* Protected routes with Navbar */}
  {auth?.isAuthenticated === true ? (
    <Route path='/' element={<NavbarLayout />}>
      <Route path='/' element={<Main />} />
      <Route path='*' element={<Navigate to='/' />} />
    </Route>
  ) : (
    <>
      <Route path='/login' element={<Login />} />
      <Route path='/register' element={<Register />} />
      <Route path='*' element={<Navigate to='/login' />} />
    </>
  )}
</Routes>
```

Şekil 5.5.1 Sayfa Yönlendirmeleri

BÖLÜM 6. UYGULAMADAN EKRAN GÖRÜNTÜLERİ

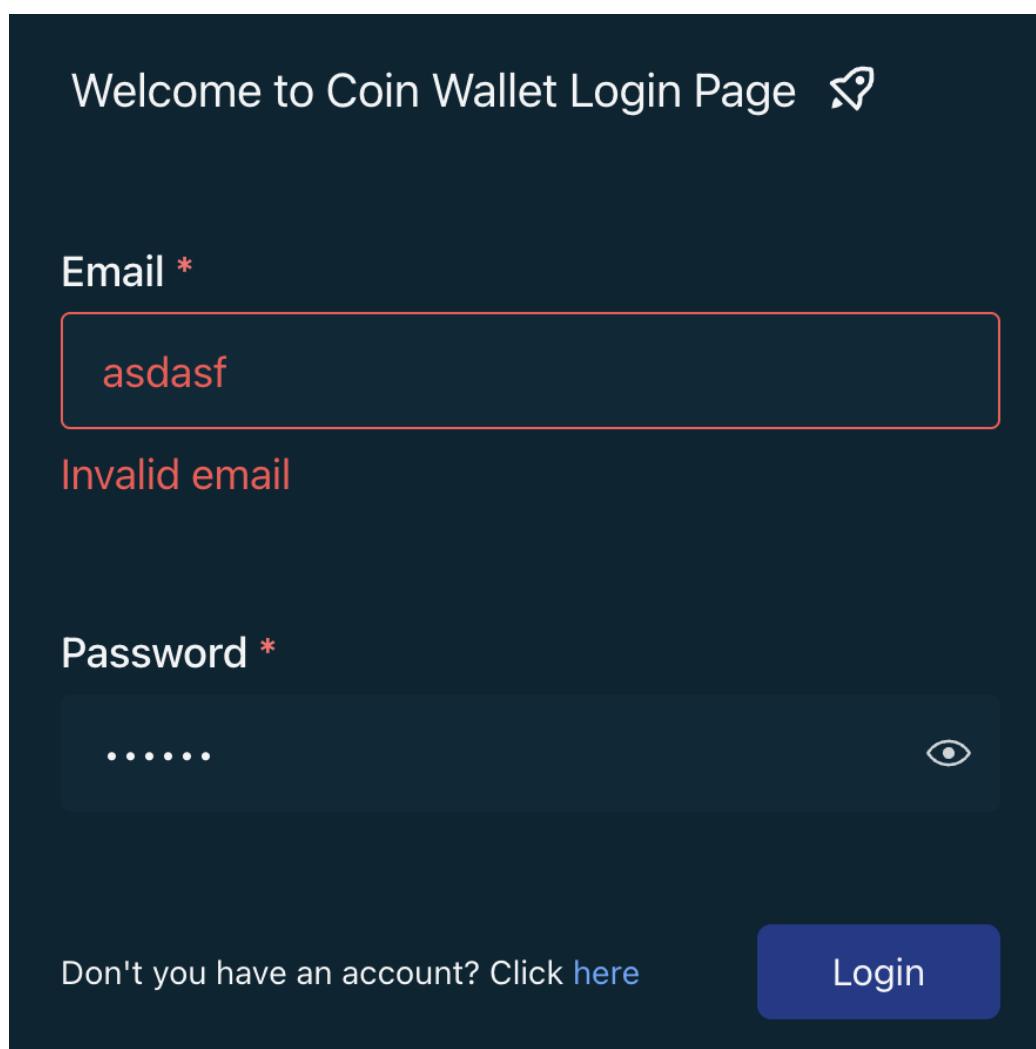
6.1. Giriş Ekranı

Giriş ekranında e-mail ve şifre girilebilmesi için 2 input alanı ve giriş yapılabilmesi için bir buton bulunmaktadır.



Şekil 6.1.1 Login Sayfası

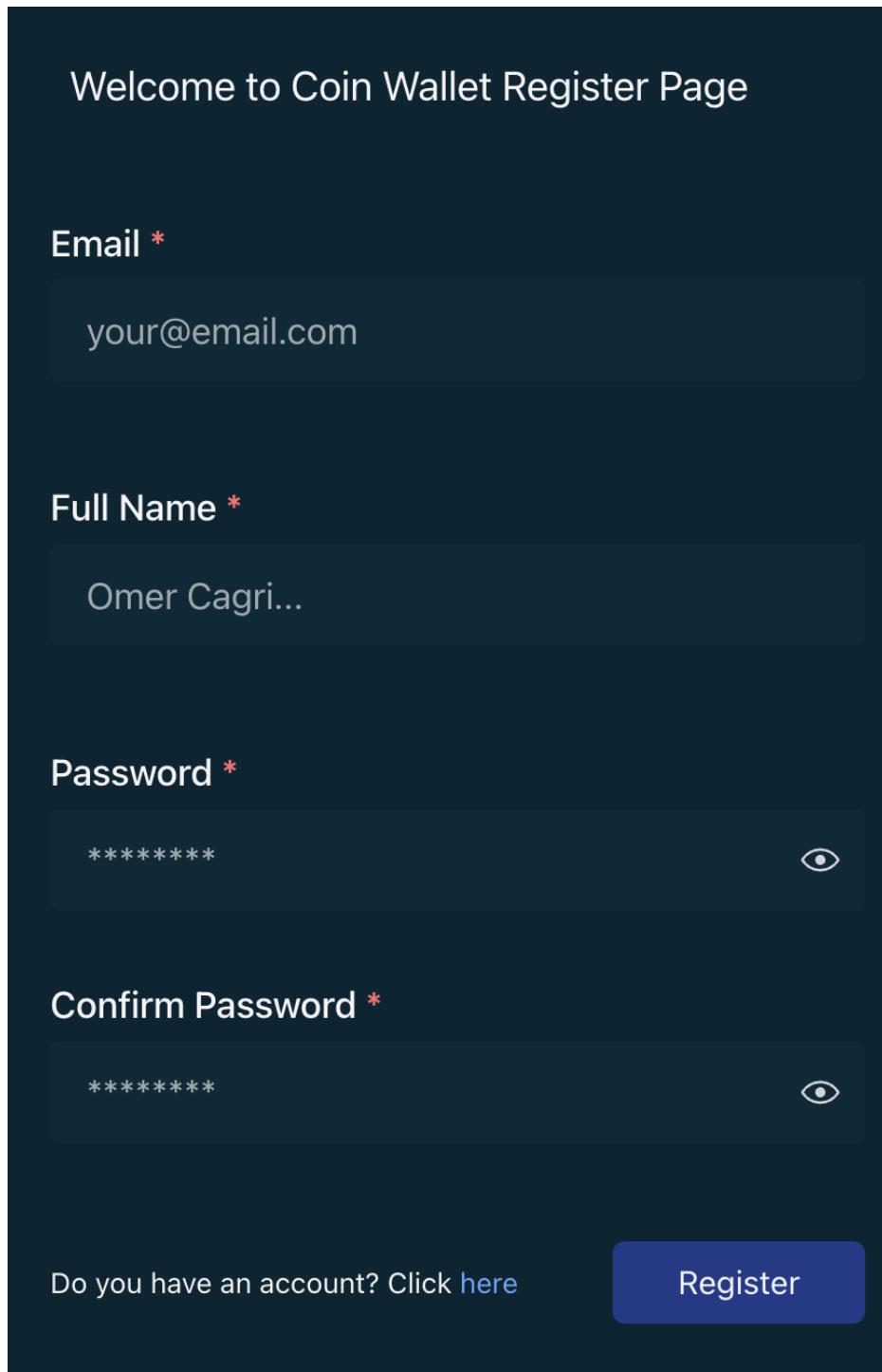
Giriş sayfasında front tarafında geçerli bir e-posta adresi girilmesini zorunlu kıلان bir validasyon bulunmaktadır.



Şekil 6.1.2 Login Sayfası geçersiz mail girişи

6.2. Kayıt Ekranı

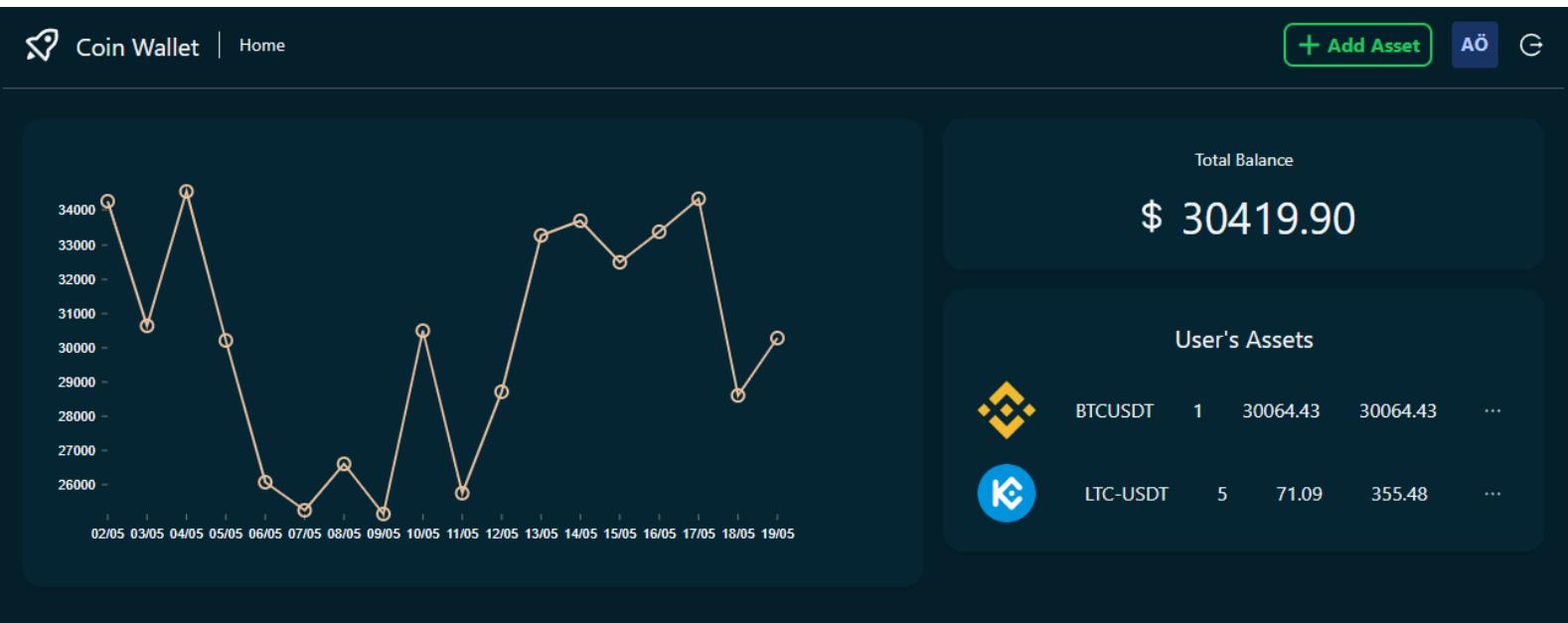
Kayıt ekranında e-posta, isim soyisim, şifre ve şifre doğrulama girilebilmesi için 4 input alanı ve kayıt işlemini gerçekleştirmek için bir buton bulunmaktadır.



Sekil 6.2.1 Register Sayfası

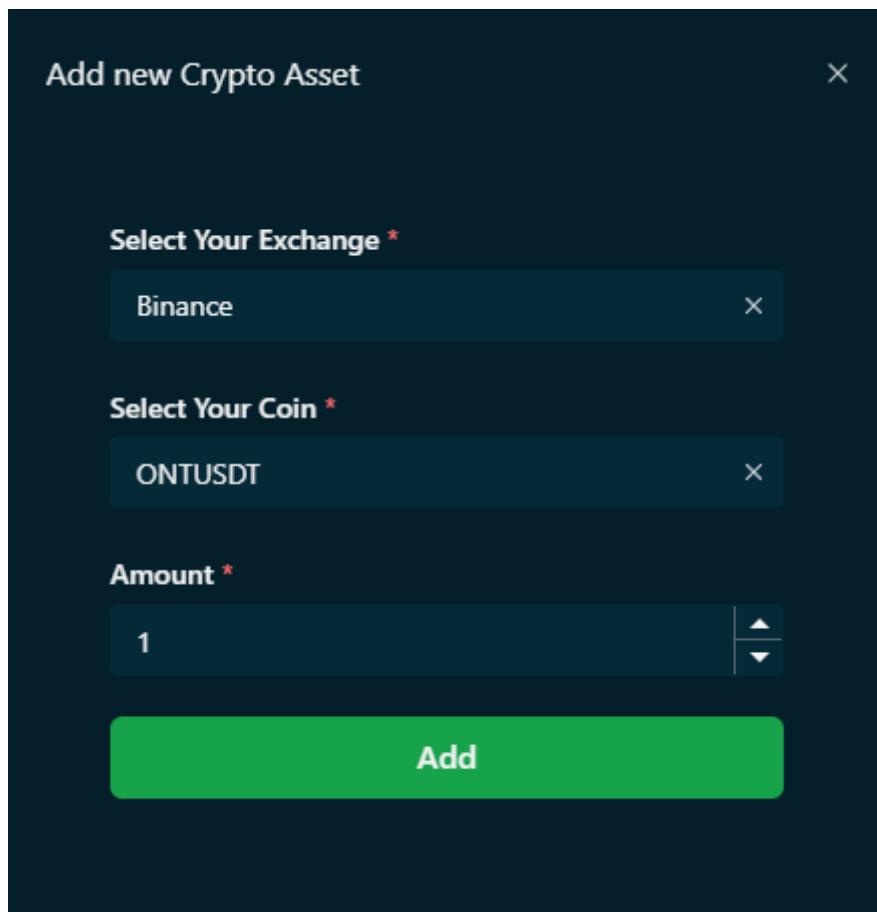
6.3. Ana Sayfa

Kullanıcı giriş yaptıktan sonra otomatik olarak dashboard sayfasına yönlendirilir. Bu sayfa basitçe 3 parçadan oluşmaktadır; cüzdanının zamana bağlı değişimini görebildiği grafik, cüzdanının toplam değerini görebileceği Total Balance kısmı ve cüzdanındaki tüm Kripto Varlıkların listelendiği User's Assets kısmı.



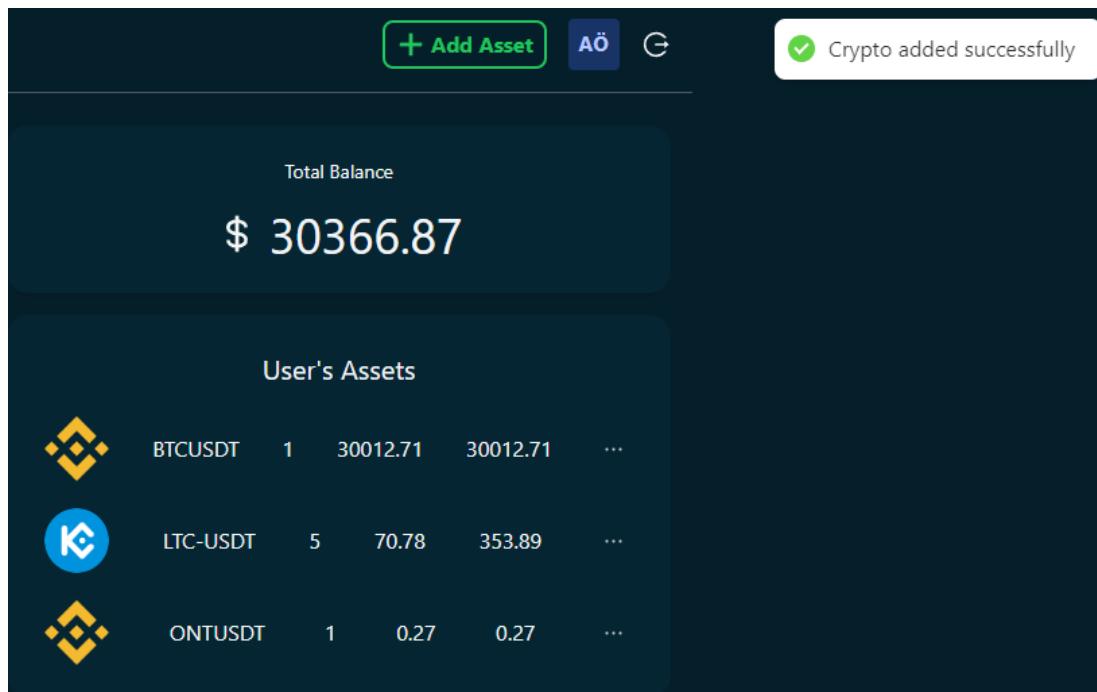
Şekil 6.3.1 Ana Sayfa

Kullanıcılar, sağ üstteki Add Asset butonuna tıklayarak Cüzdanlarına farklı Cripto Varlık Borsalarından, istenen Cripto Varlığı, istenen miktarlarda ekleyebilirler.



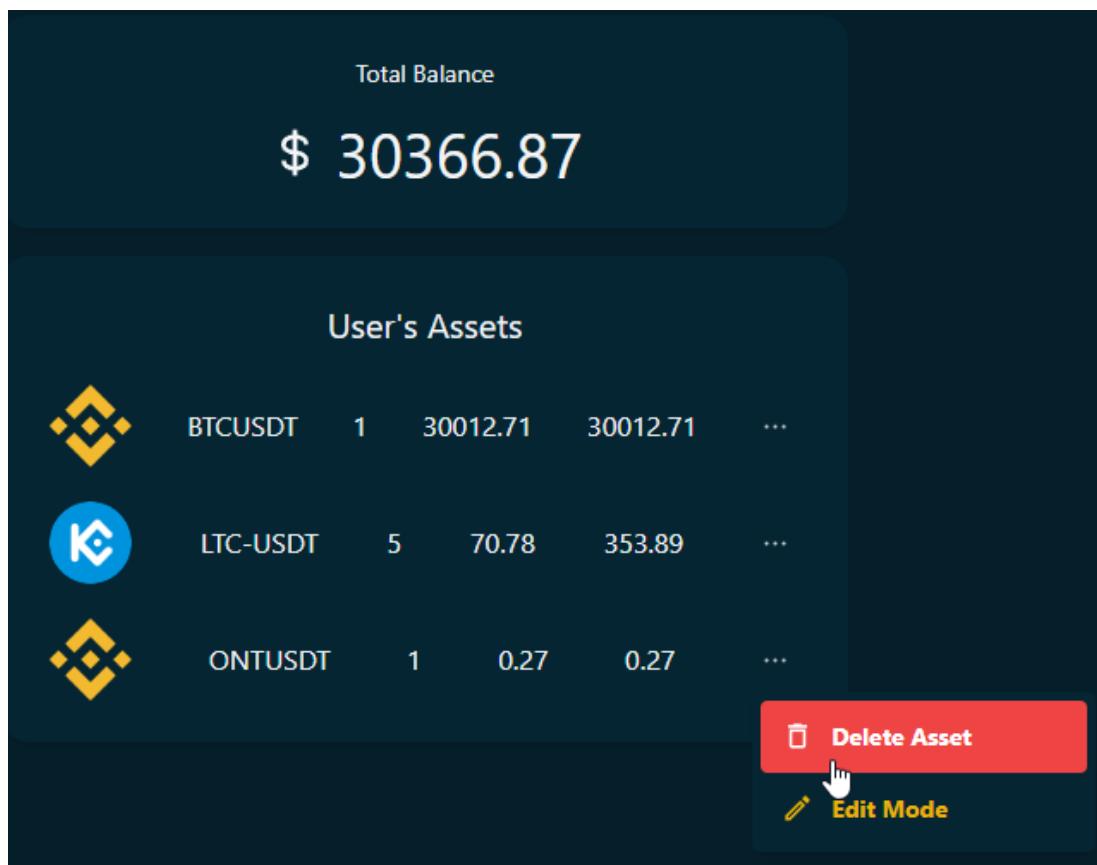
Şekil 6.3.2 Yeni Cripto Varlık ekleme ekranı

Yeni Kripto Varlık ekleme işlemi sonrası görselde görünen şekilde onay uyarısı verilir ve eklenen Kripto Varlık, User's Assets kısmına eklenir. Ekleme işlemi sonrası Total Balance değeri de uygun şekilde güncellenir.



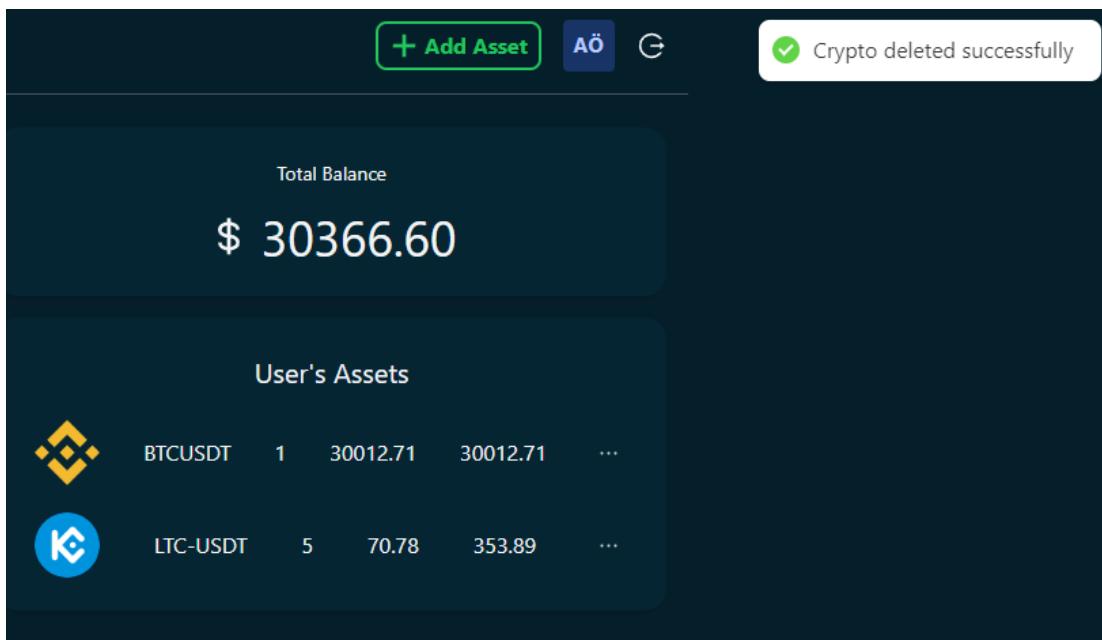
Şekil 6.3.3 Yeni Kripto Varlık eklendiğinde bildirim gelmesi

Cüzdan'a eklenmiş olan Kripto Varlıklar Cüzdan'dan silinebilmektedir. Cüzdan'da bulunan her bir Kripto Varlığın sağ tarafındaki silme butonuna tıklayıp, sonrasında çıkan onay paneline de onay verdikten sonra ilgili Kripto Varlık, Cüzdan'dan silinmektedir.



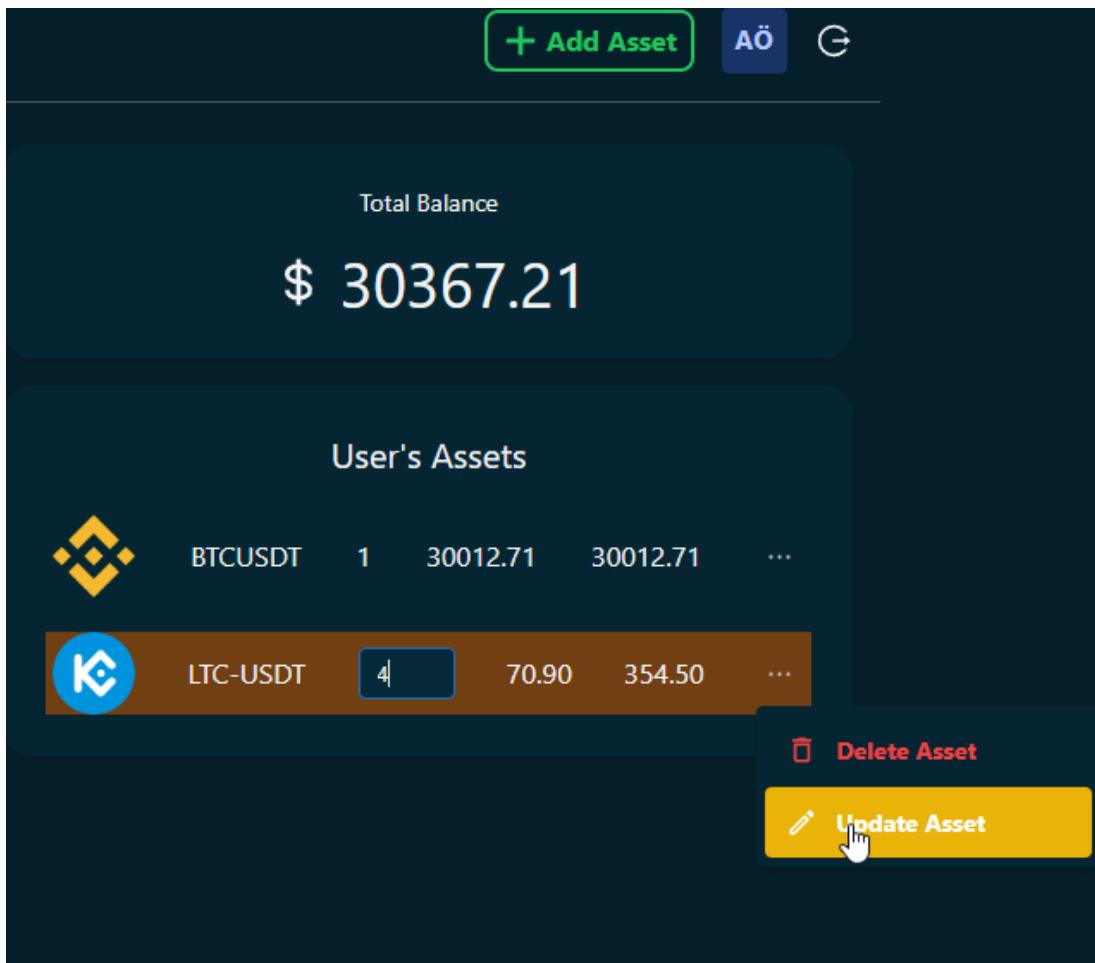
Şekil 6.3.4 Kripto varlık silme butonu

Kripto Varlık silme işlemi sonrası görselde görünen şekilde onay uyarısı verilir ve silinen Kripto Varlık, User's Assets kısmından silinir. Silme işlemi sonrası Total Balance değeri de uygun şekilde güncellenir.



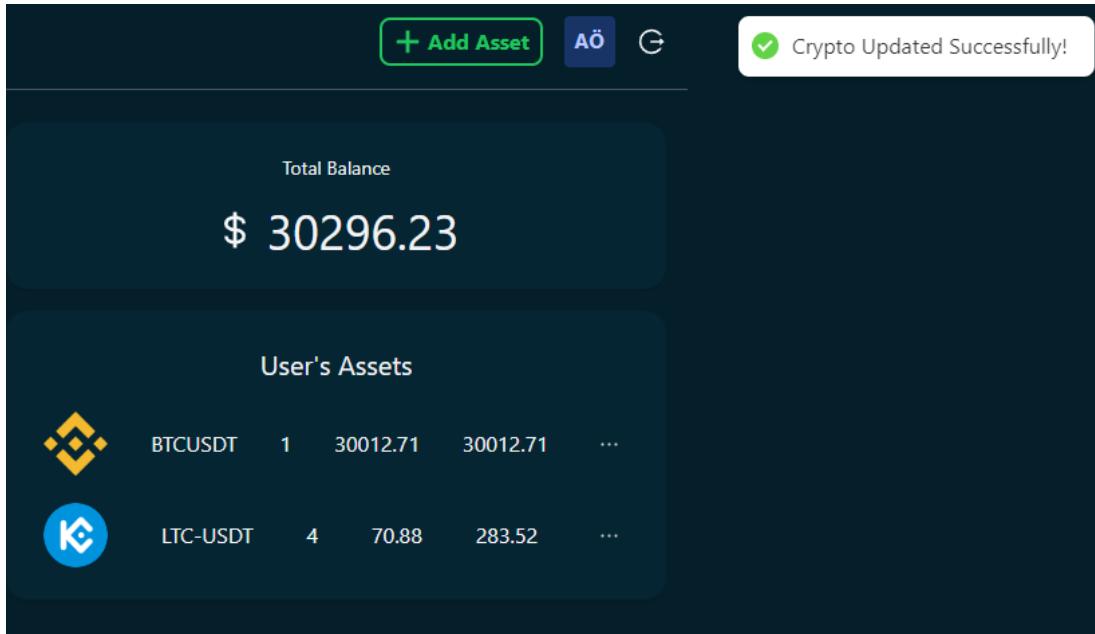
Şekil 6.3.5 Kripto Varlık silindiğinde bildirim gelmesi

Cüzdan'a eklenmiş her Kripto Varlığın miktarı güncellenebilmektedir. Cüzdan'da bulunan her bir Kripto Varlığın sağ tarafındaki Edit butonuna tıklayıp, sonrasında açılan edit mod ile ilgili Kripto Varlığın miktarı güncellenmektedir.



Şekil 6.3.6 Kripto Varlık güncelleme butonu

Kripto Varlık güncelleme işlemi sonrası görselde görünen şekilde onay uyarısı verilir ve güncellenen Kripto Varlık, User's Assets kısmında da güncellenir. Güncelleme işlemi sonrası Total Balance değeri de uygun şekilde güncellenir.



Şekil 6.3.6 Kripto Varlık güncelleme işlemi onay bildirimi

BÖLÜM 7. SONUÇLAR

Bu projede, kullanıcıların Kripto Varlıklarının bulunabileceği en büyük 3 Kripto Varlık Borsasının ikisi olan Binance ve KuCoin borsaları bulunmaktadır. Uygulamayı kullanacak olan kullanıcılar bu iki borsada olan tüm varlıkları uygulama içerisindeki Cüzdanlarına ekleyebilir ve Dashboard üzerinden kâr/zarar durumlarını takip edebilirler.

Projede kullanılan araçlar kısaca şu şekildedir;

- Backend - Node.js, Express.js, Typescript
- Job Handler - Node.js, Typescript, Cron Expression
- Veri Tabanı - NoSQL, MongoDB
- Frontend - React.js, Mantine, Typescript, TailwindCSS
- Harici Kripto Varlık Borsa API'leri - Binance, KuCoin

Hedeflenen amaca ulaşabilmek için CodeFirst yöntemi ile çalışıldı. Kullanılacak modeller öncelikle kodda tanımlanıp daha sonra veritabanında tanımlandı. Server ve Job Handler tarafları için NodeJS ve TypeScript kullanılırken Önyüz için React, TypeScript, TailwindCSS gibi teknolojiler kullanıldı.

Proje diğer uygulama ve web sitelerinin aksine farklı Kripto Para Borsalarının (Binance, KuCoin) sunduğu kripto varlıkların takip edilmesine olanak sağlamaktadır. Bu amaçla geliştirilmeye başlanan proje amacına ulaşmıştır.

Uygulamaya <https://www.coinwallet.dev/> adresinden erişilebilir.

KAYNAKLAR

- [1] <https://github.com/binance/binance-spot-api-docs/blob/master/rest-api.md>.
- [2] <https://docs.kucoin.com/#general>
- [3] https://www.youtube.com/playlist?list=PLvzuUVysUFOuB1kJQ3S2G-nB7_nHhD7Ay
- [4] <https://mantine.dev/getting-started/>
- [5] <https://tailwindcss.com/docs/installation>
- [6] <https://reactrouter.com/docs/en/v6/examples/auth>
- [7] <https://expressjs.com/en/starter/installing.html>
- [8] <https://mongoosejs.com/docs/guide.html>
- [9] <https://github.com/harrisiiarak/cron-parser>
- [10] <https://en.wikipedia.org/wiki/Cron>