Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

\_\_\_\_\_\_\_\_\_\_\_ Едуард ЖАРІКОВ

“\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2023 р.

**Веб-застосунок для відстеження цін на криптоактиви з оповіщенням про зміну ціни**

**Текст програми**

КПІ.ІТ-9104.045440.03.12

“ПОГОДЖЕНО”

Керівник проєкту:

\_\_\_\_\_\_\_\_\_\_\_\_ Валерій НОВІНСЬКИЙ

|  |  |
| --- | --- |
| Нормоконтроль: | Виконавець: |
| \_\_\_\_\_\_\_ Максим ГОЛОВЧЕНКО | \_\_\_\_\_\_ Артур-Юрій КОРЧИНСЬКИЙ |

Київ – 2023

**Файл aggregator.rs**

use csb\_db\_crypto::models::{AggregatorStatus, Cryptocurrency, NewAggregatorPrice, Price};

use futures::future::join\_all;

pub(crate) struct Aggregator {

    context: csb\_db\_crypto::Db,

    currency\_id: i32,

    process\_currencies: Vec<Cryptocurrency>,

}

impl Aggregator {

    pub(crate) fn new(

        context: csb\_db\_crypto::Db,

        currency\_id: i32,

        process\_currencies: Vec<Cryptocurrency>,

    ) -> Self {

        Self {

            context,

            currency\_id,

            process\_currencies,

        }

    }

    async fn update(

        &self,

        crypto: &Cryptocurrency,

        update\_time: chrono::NaiveDateTime,

    ) -> anyhow::Result<()> {

        let mut connection = self.context.db\_connection.get().await?;

        let prices = csb\_db\_crypto::models::Price::get\_any\_price\_between(

            &mut connection,

            crypto.id,

            self.currency\_id,

            update\_time - chrono::Duration::minutes(3),

            update\_time,

        )

        .await?;

        if prices.is\_empty() {

            return Ok(());

        }

        let median = prices[prices.len() / 2].price.clone();

        let first\_quartile = prices[prices.len() / 4].price.clone();

        let third\_quartile = prices[(prices.len() \* 3) / 4].price.clone();

        let aggregated\_price = NewAggregatorPrice::new(

            crypto.id,

            self.currency\_id,

            median,

            first\_quartile,

            third\_quartile,

            update\_time,

        );

        aggregated\_price.save(&mut connection).await?;

        Price::mark\_as\_processed(&mut connection, crypto.id, self.currency\_id, prices).await?;

        Ok(())

    }

    async fn update\_time(&self, time: chrono::NaiveDateTime) -> anyhow::Result<AggregatorStatus> {

        let mut connection = self.context.db\_connection.get().await?;

        Ok(csb\_db\_crypto::models::AggregatorStatus::update\_time(

            &mut connection,

            self.currency\_id,

            time,

        )

        .await?)

    }

    pub(crate) async fn run(&self) {

        let period = tokio::time::Duration::from\_secs(60);

        let mut interval = tokio::time::interval(period);

        let period = chrono::Duration::from\_std(period).expect("Failed to convert to chrono");

        let mut update\_time = {

            let mut connection = self

                .context

                .db\_connection

                .get()

                .await

                .expect("Failed to connect to db");

            csb\_db\_crypto::models::AggregatorStatus::by\_crypto\_id\_and\_currency\_id(

                &mut connection,

                self.currency\_id,

            )

            .await

            .map(|s| s.timestamp)

            .unwrap\_or\_else(|\_| chrono::Utc::now().naive\_utc())

        };

        loop {

            let current\_time = chrono::Utc::now().naive\_utc();

            if update\_time > current\_time {

                interval.tick().await;

            }

            let mut result = vec![];

            result.reserve(self.process\_currencies.len());

            for currency in &self.process\_currencies {

                result.push(self.update(currency, update\_time));

            }

            let result = join\_all(result).await;

            for r in result {

                if let Err(e) = r {

                    eprintln!("Failed to update cryptocurrency: {e}");

                }

            }

            if let Err(e) = self.update\_time(update\_time).await {

                eprintln!("Failed to update aggregator status: {e}");

            }

            update\_time += period;

        }

    }

}

**Файл collectors\_common\_lib.rs**

use anyhow::Context;

use csb\_db\_crypto::models::SourceCryptoMapping;

pub struct Collector {

context: csb\_db\_crypto::Db,

required\_keys: Vec<SourceCryptoMapping>,

source\_id: i32,

currency\_id: i32,

url: String,

helper: Box<dyn CollectorHelper + Send + Sync>,

}

impl Collector {

pub fn new(

context: csb\_db\_crypto::Db,

required\_keys: Vec<SourceCryptoMapping>,

currency\_id: i32,

url: String,

helper: Box<dyn CollectorHelper + Send + Sync>,

) -> anyhow::Result<Self> {

let source\_id = required\_keys

.first()

.context("Expect at least one ky")?

.source\_id;

Ok(Self {

context,

required\_keys,

source\_id,

currency\_id,

url,

helper,

})

}

async fn update(&self) -> anyhow::Result<()> {

let response = reqwest::get(&self.url).await?;

let json = response.json::<serde\_json::Value>().await?;

let mut connection = self.context.db\_connection.get().await?;

let timestamp = chrono::Utc::now().naive\_utc();

let mut error = false;

for crypto in &self.required\_keys {

let price = self

.helper

.retrieve\_price\_from\_json(&json, &crypto.source\_key);

if let Err(e) = price {

error = true;

eprintln!("Failed to retrieve price: {e}");

continue;

}

let price = csb\_db\_crypto::models::NewPrice::new(

crypto.crypto\_id,

self.source\_id,

self.currency\_id,

price.unwrap(), // Safe to unwrap because we checked for error above

timestamp,

);

if let Err(e) = price.insert(&mut connection).await {

error = true;

eprintln!("Failed to insert price: {e}: {price:?}");

}

}

if error {

Err(anyhow::anyhow!("Failed to insert price"))

} else {

Ok(())

}

}

pub async fn run(&self) {

loop {

if let Err(e) = self.update().await {

eprintln!(

"Failed to update cryptocurrencies: {e} with {}",

self.source\_id

);

}

tokio::time::sleep(tokio::time::Duration::from\_secs(60)).await;

}

}

}

pub trait CollectorHelper {

fn create\_url(&self, required\_keys: &[SourceCryptoMapping]) -> String;

fn retrieve\_price\_from\_json(

&self,

json: &serde\_json::Value,

key: &str,

) -> anyhow::Result<bigdecimal::BigDecimal>;

}

pub async fn run(

connection\_string: String,

source\_name: &str,

helper: Box<dyn CollectorHelper + Send + Sync>,

) -> anyhow::Result<()> {

let context = csb\_db\_crypto::Db::new(connection\_string)

.await

.expect("Failed to connect to the database");

let required\_keys = csb\_db\_crypto::models::SourceCryptoMapping::load\_keys\_by\_source\_name(

&mut context

.db\_connection

.get()

.await

.expect("Expected to get a connection from the pool"),

source\_name,

)

.await

.expect("Failed to load keys");

if required\_keys.is\_empty() {

anyhow::bail!("No keys found for source {}", source\_name);

}

let url = helper.create\_url(&required\_keys);

Collector::new(context, required\_keys, 1, url, helper)?

.run()

.await;

Ok(())

}

**Файл chainlink\_collector.rs**

use std::{sync::Arc, time::Duration};

use anyhow::Context as ErrorContext;

use bigdecimal::num\_bigint::BigInt;

use chrono::NaiveDateTime;

use csb\_db\_crypto::{

models::{NewPrice, SourceCryptoMapping},

Db,

};

use ethers::{

prelude::Http,

providers::{Middleware, Provider},

types::H160,

};

use futures::future::join\_all;

use reqwest::Url;

use crate::error::CollectorError;

pub type TokenId = i32;

const EVENT\_SLEEP\_DURATION: Duration = Duration::from\_secs(60);

mod chain\_link\_abi {

use ethers::prelude::abigen;

abigen!(AggregatorV3Contract, "aggregatorV3InterfaceABI.json");

}

/// The price oracle. It fetches prices from the chainlink oracle contracts on the Ethereum network.

pub struct Collector {

provider: Arc<Provider<Http>>,

db\_context: Db,

source\_id: i32,

data: Vec<(TokenId, H160)>,

}

impl Collector {

/// Creates a new price oracle with the given subscription list.

pub async fn new(

ethereum\_node: Url,

input: Vec<SourceCryptoMapping>,

db\_context: Db,

) -> Result<Self, CollectorError> {

let provider =

Provider::<Http>::try\_from(ethereum\_node.to\_string()).with\_context(|| {

format!("Couldn't connect to ethereum node under {ethereum\_node} url",)

})?;

let provider = Arc::new(provider);

let source\_id = input[0].source\_id;

let input = input.into\_iter().map(|item| async {

let item = item;

provider

.resolve\_name(&item.source\_key)

.await

.map(|address| (item.crypto\_id, address))

});

let input: Result<Vec<(TokenId, ethers::types::H160)>, \_> =

join\_all(input).await.into\_iter().collect();

let data = input?;

Ok(Collector {

provider,

data,

db\_context,

source\_id,

})

}

/// Get the price for the given token name and address from the chainlink oracle contract.

pub async fn fetch\_price(

&self,

crypto\_id: TokenId,

address: H160,

) -> Result<(), CollectorError> {

let oracle\_contract =

chain\_link\_abi::AggregatorV3Contract::new(address, self.provider.clone());

let decimals = oracle\_contract

.decimals()

.call()

.await

.context("Couldn't fetch decimals data for {crypto\_id}")?;

let (\_, latest\_price, \_, updated\_at, \_) =

oracle\_contract.latest\_round\_data().call().await?;

if latest\_price.is\_negative() {

return Err(CollectorError::NegativeLatestPrice(crypto\_id.to\_string()));

}

let updated\_at = NaiveDateTime::from\_timestamp\_opt(updated\_at.as\_u64() as i64, 0).context(

"Couldn't convert the timestamp from the chainlink oracle contract to NaiveDateTime for {crypto\_id}",

)?;

let mut connection = self.db\_context.db\_connection.get().await.context(

"Couldn't get a connection from the pool to insert the new price into the database",

)?;

let mut bytes = [0u8; 32];

latest\_price.to\_little\_endian(&mut bytes);

let latest\_price =

bigdecimal::BigDecimal::new(BigInt::from\_signed\_bytes\_le(&bytes), decimals.into());

let new\_price = NewPrice::new(crypto\_id, self.source\_id, 1, latest\_price, updated\_at);

new\_price.insert(&mut connection).await?;

Ok(())

}

/// Get the prices for all the tokens in the subscription list.

pub async fn fetch\_all(&self) -> Vec<Result<(), CollectorError>> {

let mut result = vec![];

result.reserve(self.data.len());

for (token\_id, address) in &self.data {

result.push(self.fetch\_price(\*token\_id, \*address));

}

join\_all(result).await

}

pub async fn run(&self) -> Result<(), CollectorError> {

let mut interval = tokio::time::interval(EVENT\_SLEEP\_DURATION);

loop {

interval.tick().await;

let result = self

.fetch\_all()

.await

.into\_iter()

.collect::<Result<Vec<\_>, \_>>();

if let Err(err) = result {

eprintln!("Error fetching prices: {:?}", err);

}

}

}

}

**Файл UserService.proto**

syntax = "proto3";

package coin\_sight;

import "google/protobuf/empty.proto";

import "google/protobuf/timestamp.proto";

service UserService {

rpc login(Login) returns (LoginResponse);

rpc logout(Token) returns (google.protobuf.Empty);

rpc register(Register) returns (google.protobuf.Empty);

rpc refresh\_token(Token) returns (LoginResponse);

rpc validate\_token(Token) returns (google.protobuf.Empty);

rpc add\_notification(Notification) returns (google.protobuf.Empty);

rpc notifications(Token) returns (Notifications);

rpc remove\_notification(RemoveNotification) returns (google.protobuf.Empty);

rpc edit\_notification(EditNotification) returns (google.protobuf.Empty);

}

message Login {

string email = 1;

string password = 2;

}

message Register {

string login = 1;

string email = 2;

string password = 3;

}

message Token {

string token = 1;

}

message LoginResponse {

string token = 1;

int64 expires\_at = 2;

}

message Notification {

string token = 1;

string coin\_name = 2;

string source = 3;

string change\_type = 4;

string change\_value = 5;

string current\_price = 6;

string name = 7;

}

message Notifications {

message NotificationData {

int64 id = 1;

string coin\_name = 2;

string source = 3;

string change\_type = 4;

string change\_value = 5;

string current\_price = 6;

string name = 7;

}

repeated NotificationData notifications = 1;

}

message EditNotification {

string token = 1;

int64 id = 2;

string coin\_name = 3;

string source = 4;

string change\_type = 5;

string change\_value = 6;

string current\_price = 7;

string name = 8;

}

message RemoveNotification {

string token = 1;

int64 id = 2;

}

**Файл graphql\_api\_query.rs**

use crate::types::{crypto, user};

use crate::Context;

use csb\_db\_crypto::models;

use juniper::{graphql\_object, FieldResult};

pub struct Query;

#[graphql\_object(context = Context)]

impl Query {

fn api\_version() -> &'static str {

"1.0"

}

fn crypto() -> cryptocurrency::CryptoQuery {

cryptocurrency::CryptoQuery

}

fn price() -> price::PriceQuery {

price::PriceQuery

}

fn aggregated\_price() -> aggregated\_price::AggregatedPriceQuery {

aggregated\_price::AggregatedPriceQuery

}

fn source() -> sources::SourceQuery {

sources::SourceQuery

}

fn notification() -> notifications::NotificationQuery {

notifications::NotificationQuery

}

}

mod price {

use super::\*;

pub struct PriceQuery;

#[graphql\_object(context = Context)]

impl PriceQuery {

async fn latest(

context: &Context,

crypto\_id: i32,

currency\_id: i32,

source\_id: i32,

) -> FieldResult<crypto::Price> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(crypto::Price(

models::Price::get\_latest(&mut connection, crypto\_id, currency\_id, source\_id)

.await?,

))

}

async fn history(

context: &Context,

crypto\_id: i32,

currency\_id: i32,

source\_id: i32,

limit: i32,

offset: i32,

) -> FieldResult<Vec<crypto::Price>> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(models::Price::get\_history\_paged(

&mut connection,

crypto\_id,

currency\_id,

source\_id,

limit,

offset,

)

.await?

.into\_iter()

.map(crypto::Price)

.collect())

}

}

}

mod aggregated\_price {

use super::\*;

pub struct AggregatedPriceQuery;

#[graphql\_object(context = Context)]

impl AggregatedPriceQuery {

async fn latest(

context: &Context,

crypto\_id: i32,

currency\_id: i32,

) -> FieldResult<crypto::AggregatedPrice> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(crypto::AggregatedPrice(

models::AggregatedPrice::get\_latest(&mut connection, crypto\_id, currency\_id)

.await?,

))

}

async fn history(

context: &Context,

crypto\_id: i32,

currency\_id: i32,

limit: i32,

offset: i32,

) -> FieldResult<Vec<crypto::AggregatedPrice>> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(models::AggregatedPrice::get\_history\_paged(

&mut connection,

crypto\_id,

currency\_id,

limit,

offset,

)

.await?

.into\_iter()

.map(crypto::AggregatedPrice)

.collect())

}

}

}

mod cryptocurrency {

use super::\*;

pub struct CryptoQuery;

#[graphql\_object(context = Context)]

impl CryptoQuery {

async fn by\_symbol(

context: &Context,

symbol: String,

) -> FieldResult<crypto::Cryptocurrency> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(crypto::Cryptocurrency(

models::Cryptocurrency::by\_symbol(&mut connection, &symbol).await?,

))

}

async fn by\_id(context: &Context, id: i32) -> FieldResult<crypto::Cryptocurrency> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(crypto::Cryptocurrency(

models::Cryptocurrency::by\_id(&mut connection, id).await?,

))

}

async fn search(

context: &Context,

query: String,

) -> FieldResult<Vec<crypto::Cryptocurrency>> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(

models::Cryptocurrency::search\_by\_symbol\_or\_name(&mut connection, query)

.await?

.into\_iter()

.map(crypto::Cryptocurrency)

.collect(),

)

}

async fn top(

context: &Context,

#[graphql(default = 10)] limit: i32,

) -> FieldResult<Vec<crypto::Cryptocurrency>> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(

models::Cryptocurrency::top\_cryptocurrencies(&mut connection, limit.into())

.await?

.into\_iter()

.map(crypto::Cryptocurrency)

.collect(),

)

}

async fn all(context: &Context) -> FieldResult<Vec<crypto::Cryptocurrency>> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(models::Cryptocurrency::all(&mut connection)

.await?

.into\_iter()

.map(crypto::Cryptocurrency)

.collect())

}

}

}

mod sources {

use crate::types::crypto::FullHistory;

use super::\*;

pub struct SourceQuery;

#[graphql\_object(context = Context)]

impl SourceQuery {

async fn full\_history(context: &Context, crypto\_id: i32) -> FieldResult<FullHistory> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

let sources = models::Source::all(&mut connection)

.await?

.into\_iter()

.map(crypto::Source)

.collect();

let aggregated\_prices =

models::AggregatedPrice::get\_history\_paged(&mut connection, crypto\_id, 1, 0, 500)

.await?

.into\_iter()

.map(crypto::AggregatedPrice)

.collect();

Ok(FullHistory {

aggregated\_prices,

sources,

})

}

async fn all(context: &Context) -> FieldResult<Vec<crypto::Source>> {

let mut connection = context.crypto\_db.db\_connection.get().await?;

Ok(models::Source::all(&mut connection)

.await?

.into\_iter()

.map(crypto::Source)

.collect())

}

}

}

mod notifications {

use crate::grpc\_error\_to\_field\_error;

use super::\*;

pub struct NotificationQuery;

#[graphql\_object(context = Context)]

impl NotificationQuery {

async fn all(context: &Context, token: String) -> FieldResult<Vec<user::Notification>> {

let mut service = context.user\_service.client().await?;

Ok(service

.notifications(csb\_comm::Token { token })

.await

.map\_err(grpc\_error\_to\_field\_error)?

.into\_inner()

.notifications

.into\_iter()

.map(Into::<user::Notification>::into)

.collect())

}

}

}

**Файл graphql\_api\_mutation.rs**

use juniper::{graphql\_object, FieldResult};

use crate::types::user;

use crate::Context;

pub struct Mutation;

use crate::grpc\_error\_to\_field\_error;

#[graphql\_object(context = Context)]

impl Mutation {

fn api\_version() -> &'static str {

"1.0"

}

fn users() -> FieldResult<users::UserMutation> {

Ok(users::UserMutation)

}

fn notification() -> notifications::NotificationMutation {

notifications::NotificationMutation

}

}

mod users {

use chrono::NaiveDateTime;

use csb\_comm::LoginResponse;

use super::\*;

pub struct UserMutation;

#[graphql\_object(context = Context)]

impl UserMutation {

async fn register(

login: String,

email: String,

password: String,

context: &Context,

) -> FieldResult<bool> {

let mut user\_service = context

.user\_service

.client()

.await

.map\_err(|\_| "Internal error")?;

let request = tonic::Request::new(csb\_comm::Register {

login,

email,

password,

});

user\_service

.register(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?;

Ok(true)

}

async fn login(

email: String,

password: String,

context: &Context,

) -> FieldResult<user::JWTToken> {

let mut user\_service = context.user\_service.client().await?;

let request = tonic::Request::new(csb\_comm::Login { email, password });

let LoginResponse { token, expires\_at } = user\_service

.login(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?

.into\_inner();

let token = user::JWTToken::new(

token,

NaiveDateTime::from\_timestamp\_opt(expires\_at, 0).ok\_or("Invalid timestamp")?,

);

Ok(token)

}

async fn logout(context: &Context, token: String) -> FieldResult<bool> {

let mut user\_service = context.user\_service.client().await?;

let request = tonic::Request::new(csb\_comm::Token { token });

user\_service

.logout(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?;

Ok(true)

}

async fn notifications(context: &Context, token: String) -> FieldResult<bool> {

let mut user\_service = context.user\_service.client().await?;

let request = tonic::Request::new(csb\_comm::Token { token });

user\_service

.validate\_token(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?;

Ok(true)

}

}

}

mod notifications {

use super::\*;

pub struct NotificationMutation;

#[graphql\_object(context = Context)]

impl NotificationMutation {

#[allow(clippy::too\_many\_arguments)]

async fn create(

context: &Context,

token: String,

name: String,

coin\_name: String,

source: String,

change\_type: String,

change\_value: String,

current\_price: String,

) -> FieldResult<bool> {

let mut user\_service = context.user\_service.client().await?;

let request = tonic::Request::new(csb\_comm::Notification {

token,

coin\_name,

source,

change\_type,

change\_value,

current\_price,

name,

});

user\_service

.add\_notification(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?;

Ok(true)

}

async fn remove(context: &Context, token: String, id: i32) -> FieldResult<bool> {

let mut user\_service = context.user\_service.client().await?;

let request = tonic::Request::new(csb\_comm::RemoveNotification {

token,

id: id as i64,

});

user\_service

.remove\_notification(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?;

Ok(true)

}

#[allow(clippy::too\_many\_arguments)]

async fn edit(

context: &Context,

token: String,

id: i32,

name: String,

coin\_name: String,

source: String,

change\_type: String,

change\_value: String,

current\_price: String,

) -> FieldResult<bool> {

let mut user\_service = context.user\_service.client().await?;

let request = tonic::Request::new(csb\_comm::EditNotification {

token,

id: id as i64,

coin\_name,

source,

change\_type,

change\_value,

current\_price,

name,

});

user\_service

.edit\_notification(request)

.await

.map\_err(grpc\_error\_to\_field\_error)?;

Ok(true)

}

}

}

**Файл UserService.rs**

use std::str::FromStr;

use csb\_comm::{

notifications::NotificationData, EditNotification, Login, LoginResponse, Notification,

Notifications, Register, RemoveNotification, Token,

};

use csb\_db\_user::AsyncPgConnection;

use hmac::digest::KeyInit;

use hmac::Hmac;

use jwt::{SignWithKey, VerifyWithKey};

use rand::Rng;

use tonic::{Request, Response, Status};

#[derive(serde::Serialize, serde::Deserialize)]

struct Claims {

user\_id: i32,

exp: i64,

}

pub struct UserService {

context: csb\_db\_user::Db,

key: hmac::Hmac<sha2::Sha256>,

}

impl UserService {

pub fn new(context: csb\_db\_user::Db, key: Vec<u8>) -> Self {

let key = Hmac::new\_from\_slice(&key).expect("HMAC can take key of any size");

Self { context, key }

}

}

#[tonic::async\_trait]

impl csb\_comm::user\_service\_server::UserService for UserService {

async fn login(&self, request: Request<Login>) -> Result<Response<LoginResponse>, Status> {

let Login { email, password } = request.into\_inner();

let mut connection = self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

let user = csb\_db\_user::models::User::by\_email(&mut connection, &email)

.await

.map\_err(|\_| Status::internal("Database failure"))?

.ok\_or(Status::not\_found("Incorrect email/password"))?;

let result = argon2::verify\_encoded(&user.password\_hash, password.as\_bytes())

.map\_err(|\_| Status::internal("Error while verifying password hash"))?;

if !result {

return Err(Status::unauthenticated("Incorrect email/password"));

}

let login\_response = create\_token\_for\_user(user.id, &self.key, &mut connection).await?;

Ok(Response::new(login\_response))

}

async fn register(&self, request: Request<Register>) -> Result<Response<()>, Status> {

let Register {

login,

email,

password,

} = request.into\_inner();

if !check\_if\_mail\_is\_valid(&email) {

return Err(Status::invalid\_argument("Invalid email"));

}

if password.len() < 8 {

return Err(Status::invalid\_argument("Password too short"));

}

let random\_salt = rand::thread\_rng().gen::<[u8; 16]>();

let password\_hash =

argon2::hash\_encoded(password.as\_bytes(), &random\_salt, &Default::default())

.map\_err(|\_| Status::internal("Error while hashing password"))?;

let new\_user = csb\_db\_user::models::NewUser {

login,

email,

password\_hash,

default\_notification\_method: "".to\_string(),

};

let mut connection = self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

if csb\_db\_user::models::User::by\_email(&mut connection, &new\_user.email)

.await

.map\_err(|\_| Status::internal("Database failure"))?

.is\_some()

{

return Err(Status::already\_exists(

"User with this email already exists",

));

}

if csb\_db\_user::models::User::by\_login(&mut connection, &new\_user.login)

.await

.map\_err(|\_| Status::internal("Database failure"))?

.is\_some()

{

return Err(Status::already\_exists(

"User with this login already exists",

));

}

let updated\_rows = new\_user

.insert(&mut connection)

.await

.map\_err(|\_| Status::internal("Failed to insert into database"))?;

if updated\_rows == 0 {

return Err(Status::internal("Failed to insert into database"));

}

Ok(tonic::Response::new(()))

}

async fn logout(&self, request: Request<Token>) -> Result<Response<()>, Status> {

let token = request.into\_inner().token;

let claims: Claims = token

.verify\_with\_key(&self.key)

.map\_err(|\_| Status::unauthenticated("Invalid token"))?;

let mut connection = self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

csb\_db\_user::models::Token::remove\_user\_token(&mut connection, claims.user\_id, &token)

.await

.map\_err(|\_| Status::internal("Error while removing token from the database"))?;

Ok(Response::new(()))

}

async fn refresh\_token(

&self,

request: Request<Token>,

) -> Result<Response<LoginResponse>, Status> {

let token = request.into\_inner().token;

let claims = self.validate(token.clone()).await?; // Validate token first

let mut connection = self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?; // Get connection from the pool

let login\_response =

create\_token\_for\_user(claims.user\_id, &self.key, &mut connection).await?;

self.logout(Request::new(Token {

token: token.clone(),

}))

.await?; // Revoke old token

Ok(Response::new(login\_response))

}

async fn validate\_token(&self, request: Request<Token>) -> Result<Response<()>, Status> {

let token = request.into\_inner().token;

self.validate(token).await?;

Ok(Response::new(()))

}

async fn add\_notification(

&self,

request: Request<Notification>,

) -> Result<Response<()>, Status> {

let Notification {

token,

coin\_name,

source,

change\_type,

change\_value,

current\_price,

name,

} = request.into\_inner();

let Claims { user\_id, .. } = self.validate(token).await?;

let db\_connection = &mut self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

let current\_price = bigdecimal::BigDecimal::from\_str(&current\_price)

.map\_err(|\_| Status::invalid\_argument("Invalid current price"))?;

let change\_value = bigdecimal::BigDecimal::from\_str(&change\_value)

.map\_err(|\_| Status::invalid\_argument("Invalid change value"))?;

let (type\_, value\_change, percent\_change) = match change\_type.as\_str() {

"by Value" => (change\_type, Some(change\_value), None),

"by Percent" => (change\_type, None, Some(change\_value)),

\_ => return Err(Status::invalid\_argument("Invalid change type")),

};

let notification = csb\_db\_user::models::NewNotification {

name,

user\_id,

cryptocurrency: coin\_name,

source,

value\_change,

percent\_change,

type\_,

current\_price,

};

notification.insert(db\_connection).await.map\_err(|\_| {

Status::internal("Error. Failed to insert notification into the database")

})?;

Ok(Response::new(()))

}

async fn remove\_notification(

&self,

request: Request<RemoveNotification>,

) -> Result<Response<()>, Status> {

let RemoveNotification { token, id } = request.into\_inner();

let Claims { user\_id, .. } = self.validate(token).await?;

let db\_connection = &mut self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

let updated =

csb\_db\_user::models::Notification::remove\_by\_id(id as i32, user\_id, db\_connection)

.await

.map\_err(|\_| {

Status::internal("Error. Failed to remove notification from the database")

})?;

if updated == 0 {

return Err(Status::not\_found("Notification not found"));

}

Ok(Response::new(()))

}

async fn notifications(

&self,

request: Request<Token>,

) -> Result<Response<Notifications>, Status> {

let token = request.into\_inner().token;

let Claims { user\_id, .. } = self.validate(token).await?;

let db\_connection = &mut self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

let notifications =

csb\_db\_user::models::Notification::all\_from\_user(user\_id, db\_connection)

.await

.map\_err(|\_| {

Status::internal("Error. Failed to get notifications from the database")

})?;

let notifications = notifications

.into\_iter()

.map(|notification| NotificationData {

id: notification.id as i64,

coin\_name: notification.cryptocurrency,

source: notification.source,

change\_type: notification.type\_,

change\_value: notification

.value\_change

.map(|value| value.to\_string())

.unwrap\_or\_else(|| notification.percent\_change.unwrap().to\_string()),

current\_price: notification.current\_price.to\_string(),

name: notification.name,

})

.collect();

Ok(Response::new(Notifications { notifications }))

}

async fn edit\_notification(

&self,

request: Request<EditNotification>,

) -> Result<Response<()>, Status> {

let EditNotification {

token,

id,

coin\_name,

source,

change\_type,

change\_value,

current\_price,

name,

} = request.into\_inner();

let Claims { user\_id, .. } = self.validate(token).await?;

let db\_connection = &mut self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

let current\_price = bigdecimal::BigDecimal::from\_str(&current\_price)

.map\_err(|\_| Status::invalid\_argument("Invalid current price"))?;

let change\_value = bigdecimal::BigDecimal::from\_str(&change\_value)

.map\_err(|\_| Status::invalid\_argument("Invalid change value."))?;

let (type\_, value\_change, percent\_change) = match change\_type.as\_str() {

"by Value" => (change\_type, Some(change\_value), None),

"by Percent" => (change\_type, None, Some(change\_value)),

\_ => return Err(Status::invalid\_argument("Invalid change type")),

};

let mut notification =

csb\_db\_user::models::Notification::by\_id(id as i32, user\_id, db\_connection)

.await

.map\_err(|\_| Status::not\_found("Notification not found"))?;

notification.cryptocurrency = coin\_name;

notification.source = source;

notification.type\_ = type\_;

notification.value\_change = value\_change;

notification.percent\_change = percent\_change;

notification.current\_price = current\_price;

notification.name = name;

notification

.update(db\_connection)

.await

.map\_err(|\_| Status::internal("Error. Failed to update notification"))?;

Ok(Response::new(()))

}

}

impl UserService {

async fn validate(&self, token: String) -> Result<Claims, Status> {

let claims: Claims = token

.verify\_with\_key(&self.key)

.map\_err(|\_| Status::unauthenticated("Invalid token"))?;

let mut connection = self

.context

.db\_connection

.get()

.await

.map\_err(|\_| Status::internal("Error while getting connection from the pool"))?;

let now = chrono::Utc::now().naive\_utc();

if now.timestamp() > claims.exp {

csb\_db\_user::models::Token::remove\_outdated(&mut connection, now)

.await

.map\_err(|\_| Status::internal("Error while removing outdated tokens"))?;

return Err(Status::unauthenticated("Token expired"));

}

let token\_valid =

csb\_db\_user::models::Token::by\_user\_token(&mut connection, claims.user\_id, &token)

.await

.map\_err(|\_| Status::internal("Error while getting token from the database"))?

.is\_some();

if !token\_valid {

return Err(Status::unauthenticated("Revoked token"));

}

Ok(claims)

}

}

async fn create\_token\_for\_user(

user\_id: i32,

key: &Hmac<sha2::Sha256>,

connection: &mut AsyncPgConnection,

) -> Result<LoginResponse, Status> {

let now = chrono::Utc::now().naive\_utc();

let expiration = now + chrono::Duration::days(1);

let claims = Claims {

user\_id,

exp: expiration.timestamp(),

};

let token\_str = claims

.sign\_with\_key(key)

.map\_err(|\_| Status::internal("Error while signing token"))?;

let new\_token = csb\_db\_user::models::NewToken {

user\_id,

token: token\_str.clone(),

created\_at: now,

expires\_at: expiration,

};

new\_token

.insert(connection)

.await

.map\_err(|\_| Status::internal("Error while inserting token into the database"))?;

Ok(LoginResponse {

token: token\_str,

expires\_at: expiration.timestamp(),

})

}

fn check\_if\_mail\_is\_valid(email: &str) -> bool {

let index = email.find('@');

match index {

None => false,

Some(index) if index == 0 => false,

Some(index) if index == email.len() - 1 => false,

Some(\_) => true,

}

}

**Файл NotificationService.rs**

use std::collections::HashMap;

use bigdecimal::BigDecimal;

use csb\_db\_crypto::models::{AggregatedPrice, Cryptocurrency, Price, Source};

use csb\_db\_user::models::{Notification, User};

use reqwest::header::HeaderMap;

use serde\_json::json;

enum NotificationType {

PriceChange,

PercentChange,

}

impl NotificationType {

fn from\_str(notification\_type: &str) -> Option<Self> {

match notification\_type {

"by Value" => Some(Self::PriceChange),

"by Percent" => Some(Self::PercentChange),

\_ => None,

}

}

}

enum SourceType {

Source(i32),

Median,

Q1,

Q3,

}

impl SourceType {

fn from\_str(source: &str) -> Option<Self> {

match source {

"Median Price" => Some(Self::Median),

"First Quartile Price" => Some(Self::Q1),

"Third Quartile Price" => Some(Self::Q3),

\_ => None,

}

}

}

struct ParsedNotification {

user\_id: i32,

crypto\_id: i32,

notification\_type: NotificationType,

source: SourceType,

value: BigDecimal,

price: BigDecimal,

name: String,

}

impl ParsedNotification {

async fn verify\_applicable(&self, crypto\_db: &csb\_db\_crypto::Db) -> Option<bool> {

let mut connection = crypto\_db.db\_connection.get().await.ok()?;

let price = match self.source {

SourceType::Source(source\_id) => {

Price::get\_latest(&mut connection, self.crypto\_id, 1, source\_id)

.await

.ok()?

.price

}

SourceType::Median => {

AggregatedPrice::get\_latest(&mut connection, self.crypto\_id, 1)

.await

.ok()?

.median\_price

}

SourceType::Q1 => {

AggregatedPrice::get\_latest(&mut connection, self.crypto\_id, 1)

.await

.ok()?

.first\_quartile\_price

}

SourceType::Q3 => {

AggregatedPrice::get\_latest(&mut connection, self.crypto\_id, 1)

.await

.ok()?

.third\_quartile\_price

}

};

let diff = (price - self.price.clone()).abs();

Some(match self.notification\_type {

NotificationType::PriceChange => diff > self.value,

NotificationType::PercentChange => {

let percent\_change = diff / self.price.clone() \* BigDecimal::from(100);

self.value > percent\_change

}

})

}

}

pub(crate) struct Notifier {

user\_db: csb\_db\_user::Db,

crypto\_db: csb\_db\_crypto::Db,

}

impl Notifier {

pub(crate) fn new(

user\_db: csb\_db\_user::Db,

crypto\_db: csb\_db\_crypto::Db,

brello\_api\_key: String,

) -> Self {

Self {

user\_db,

crypto\_db,

}

}

pub(crate) async fn run(&mut self) {

let mut interval = tokio::time::interval(std::time::Duration::from\_secs(60));

loop {

interval.tick().await;

let user\_connection = self.user\_db.db\_connection.get().await;

let crypto\_connection = self.crypto\_db.db\_connection.get().await;

if user\_connection.is\_err() || crypto\_connection.is\_err() {

continue;

}

let mut user\_connection = user\_connection.unwrap();

let mut crypto\_connection = crypto\_connection.unwrap();

let user\_notifications =

csb\_db\_user::models::Notification::all(&mut user\_connection).await;

let latest\_crypto\_data =

csb\_db\_crypto::models::Cryptocurrency::all(&mut crypto\_connection).await;

let sources = csb\_db\_crypto::models::Source::all(&mut crypto\_connection).await;

if user\_notifications.is\_err() || latest\_crypto\_data.is\_err() || sources.is\_err() {

continue;

}

drop(user\_connection);

drop(crypto\_connection);

self.process\_notifications(

user\_notifications.unwrap(),

latest\_crypto\_data.unwrap(),

sources.unwrap(),

)

.await;

}

}

async fn process\_notifications(

&mut self,

notifications: Vec<Notification>,

crypto: Vec<Cryptocurrency>,

sources: Vec<Source>,

) {

let crypto: HashMap<String, i32> = crypto.into\_iter().map(|c| (c.name, c.id)).collect();

let sources: HashMap<String, i32> = sources.into\_iter().map(|s| (s.name, s.id)).collect();

for notification in notifications {

let parsed\_notification = self.parse\_notification(&crypto, &sources, notification);

if parsed\_notification.is\_none() {

continue;

}

let parsed\_notification = parsed\_notification.unwrap();

if let Some(true) = parsed\_notification.verify\_applicable(&self.crypto\_db).await {

println!(

"Sending notification to user {} for notification {}",

parsed\_notification.user\_id, parsed\_notification.name

);

let connection = self.user\_db.db\_connection.get().await;

if connection.is\_err() {

println!("Failed to get connection to user db");

continue;

}

let user = csb\_db\_user::models::User::by\_id(

&mut connection.unwrap(),

parsed\_notification.user\_id,

)

.await;

if user.is\_err() {

println!("Failed to get user {}", parsed\_notification.user\_id);

continue;

}

let user = user.unwrap();

if user.is\_none() {

println!("User {} does not exist", parsed\_notification.user\_id);

continue;

}

if send\_notification(&parsed\_notification, &user.unwrap())

.await

.is\_none()

{

println!(

"Failed to send notification to user {}",

parsed\_notification.user\_id

);

}

}

}

}

fn parse\_notification(

&self,

crypto: &HashMap<String, i32>,

sources: &HashMap<String, i32>,

notification: Notification,

) -> Option<ParsedNotification> {

let crypto\_id = \*crypto.get(&notification.cryptocurrency)?;

let source\_id = if let Some(id) = sources.get(&notification.source) {

SourceType::Source(\*id)

} else {

SourceType::from\_str(&notification.source)?

};

let notification\_type = NotificationType::from\_str(&notification.type\_)?;

let value = match notification\_type {

NotificationType::PriceChange => notification.value\_change?,

NotificationType::PercentChange => notification.percent\_change?,

};

Some(ParsedNotification {

user\_id: notification.user\_id,

crypto\_id,

notification\_type,

source: source\_id,

value,

name: notification.name.clone(),

price: notification.current\_price,

})

}

}