САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 2

Выполнила:

Афанасьева Ирина Максимовна

Группа:

К33402

Проверил:
Добряков Д. И.

Санкт-Петербург

2023 г.

Задание

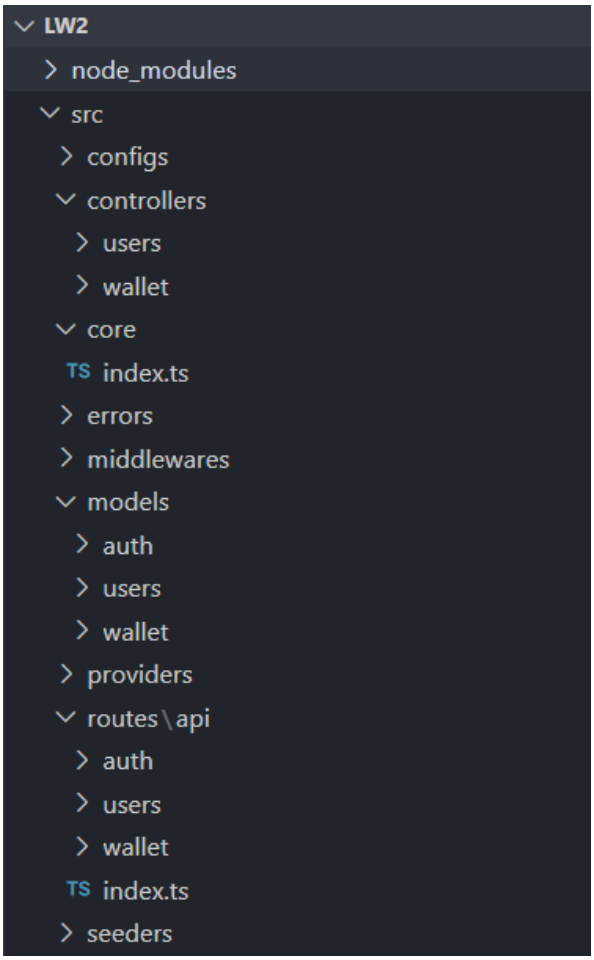
По выбранному варианту необходимо реализовать RESTful API средствами express + typescript (используя ранее написанный boilerplate).

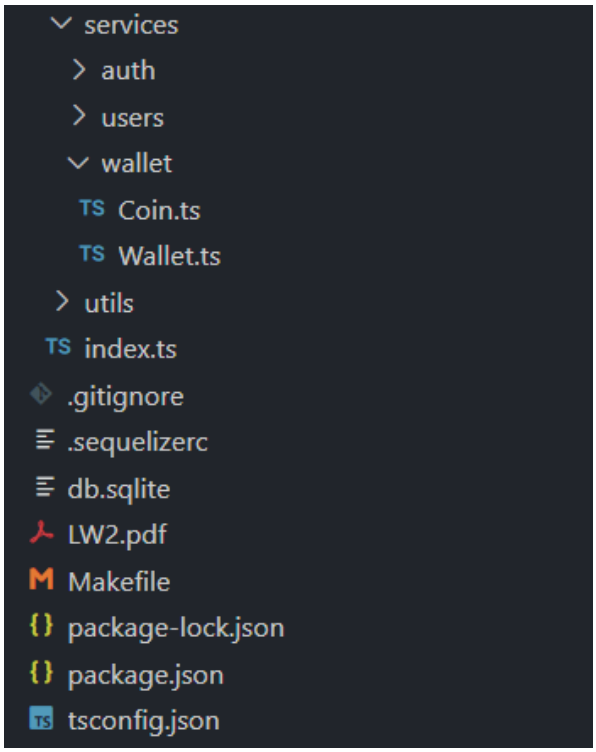
Ход работы

Выбранный вариант: сайт криптобиржи

- Вход
- Регистрация
- Портфель пользователя с указанием различных криптовалют и их количеством
- Графики роста криптовалют
- Поиск по криптовалютам с возможностью фильтрации по дате добавления на биржу

Структура приложения:





Модели:

RefreshToken.ts

```
@Table
class RefreshToken extends Model {
    @Unique
    @AllowNull(false)
    @Column
    token: string

    @ForeignKey(() => User)
    @Column
    userId: number
}

export default RefreshToken
```

Coin.ts

```
@Table
class Coin extends Model {
    @PrimaryKey
    @AllowNull(false)
    @Column
    ticker: string

    @AllowNull(false)
    @Column
    name: string

    @BelongsToMany(() => Wallet, () => CoinWallet)
    wallets: Wallet[]
}

export default Coin
```

User.ts

```ts
@Table
class User extends Model {
    @AllowNull(false)
    @Column
    firstName: string

    @AllowNull(false)
    @Column
    lastName: string

    @AllowNull(false)
    @Unique
    @Column
    email: string

    @HasMany(() => Wallet)
    wallets: [Wallet]

    @AllowNull(false)
    @Column
    password: string

    @Column
    salt: string

    @BeforeCreate
    static setPasswordInitial(instance: User) {
        instance.salt = generateSalt()
        const {password} = instance
        instance.password = encodePassword(password, instance.salt)
    }

    @BeforeUpdate
    static setPasswordUpdate(instance: User) {
        const {password} = instance
        if (instance.changed('password')) {
            instance.password = encodePassword(password, instance.salt)
```

CoinWallet.ts

```ts
@Table
class CoinWallet extends Model {
    @AllowNull(false)
    @Column
    amount: number

    @ForeignKey(() => Coin)
    @Column
    coinId: number

    @ForeignKey(() => Wallet)
    @Column
    walletId: number
}
```

Wallet.ts

```
@Table
class Wallet extends Model {
    @AllowNull(false)
    @Column
    name: string

    @Default(0)
    @Column({
        type: DataType.FLOAT,
        set(this, val): void {
            console.log('Balance not allowed to be set')
        }
    })
    balance: number

    @ForeignKey(() => User)
    @Column
    userId: number

    @BelongsTo(() => User)
    user: User

    @HasMany(() => CoinWallet)
    coinWallets: CoinWallet[]

    @BelongsToMany(() => Coin, () => CoinWallet)
    coins: Coin[]
}
```

Новые контроллеры:

Coin.ts

```
class CoinController {
    private coinService: CoinService

    constructor() {
        this.coinService = new CoinService()
    }

    get = async (request: any, response: any) => {
        const {ticker} = request.params
        try {
            response.send(await this.coinService.getByTicker(ticker))
        } catch (e: any) {
            response.status(404).send({'detail': e.message})
        }
    }
```

```
getAll = async (request: any, response: any) => {
    const time = request.query.time
    let seconds = null
    if (time) {
        seconds = TIME_INTERVAL_MAPPING[time]
    }
    if (seconds) {
        response.send(await this.coinService.getAllFilter(new Date(Date.now() - seconds * 1000)))
    } else {
        response.send(await this.coinService.getAll())
    }
}

price = async (request: any, response: any) => {
    const {ticker} = request.params
    try {
        response.send(await this.coinService.getPrice(ticker))
    } catch (e: any) {
        response.status(404).send({'detail': e.message})
    }
```

Wallet.ts

```
class WalletController {
    private walletService: WalletService

    constructor() {
        this.walletService = new WalletService()
    }

    private wrapperFunc = async (
        request: any,
        response: any,
        func: (id: number,walletData: any) => Promise<Wallet | APIError>
    ) => {

        const {body, user} = request
        const {id} = request.params
        if (user && await this.walletService.checkUser(id, user.id)) {
            try {
                const wallet = await func(id, body)
                response.status(200).send(wallet)
            } catch (error: any) {
                response.status(400).send({'detail': error.message})
            }
        } else {
            response.status(403).send({'detail': 'Unauthorized'})
        }
    }
```

```
get = async (request: any, response: any) => {
    const {user} = request
    const {id} = request.params
    if (user && await this.walletService.checkUser(id, user.id)) {
        try {
            const wallet: Wallet | APIError = await this.walletService.getById(id)
            response.status(200).send(wallet)
        } catch (error: any) {
            response.status(404).send({'detail': error.message})
        }
    } else {
        response.status(403).send({'detail': 'Unauthorized'})
    }
}

getAll = async (request: any, response: any) => {
    const {user} = request
    if (user) {
        response.send(await this.walletService.getByUserId(user.id))
    } else {
        response.status(403).send({'detail': 'Unauthorized'})
    }
}
```

```
post = async (request: any, response: any) => {
    const {body, user} = request
    if (user) {
        try {
            const wallet: Wallet | APIError = await this.walletService.create(body, user)
            response.status(201).send(wallet)
        } catch (error: any) {
            response.status(400).send({'detail': error.message})
        }
    } else {
        response.status(401).send({'detail': 'Not authenticated'})
    }
}

increaseBalance = async (request: any, response: any) => {
    return this.wrapperFunc(request, response, this.walletService.increaseBalance)
}

decreaseBalance = async (request: any, response: any) => {
    return this.wrapperFunc(request, response, this.walletService.decreaseBalance)
}

buyCoin = async (request: any, response: any) => {
    return this.wrapperFunc(request, response, this.walletService.buyCoin)
}

sellCoin = async (request: any, response: any) => {
    return this.wrapperFunc(request, response, this.walletService.sellCoin)
}

sellAllCoin = async (request: any, response: any) => {
    return this.wrapperFunc(request, response, this.walletService.sellAllCoin)
}
```

Новые сервисы:

Coin.ts

```ts
class CoinService {
    async getByTicker(ticker: string): Promise<Coin | APIError> {
        const coin = await Coin.findByPk(ticker)
        if (coin) {
            return coin.toJSON()
        }
        throw new APIError('Coin not found')
    }

    async getAll(): Promise<Coin[]> {
        return await Coin.findAll()
    }

    async getAllFilter(from: Date): Promise<Coin[]> {
        return await Coin.findAll({
            where: {
                createdAt: {[Op.gte]: from}
            }
        })
    }

    async getPrice(ticker: string): Promise<{ price: number } | APIError> {
        // Get only to check
        const coin = await Coin.findByPk(ticker)
        if (coin) {
            return {price: await getCurrentPrice(ticker)}
        }
        throw new APIError('Coin not found')
    }
}
```

Wallet.ts

```ts
class WalletService {
    async getById(id: number): Promise<Wallet | APIError> {
        const wallet = await Wallet.scope('nested').findByPk(id)
        if (wallet) {
            return wallet.toJSON()
        }
        throw new APIError('Wallet not found')
    }

    async getByUserId(userId: number): Promise<Wallet[]> {
        return await Wallet.findAll({where: {userId: userId}})
    }

    async checkUser(id: number, userId: number): Promise<boolean> {
        const wallet = await Wallet.findByPk(id)
        if (wallet) {
            return wallet.userId == userId
        }
        return false
    }
```

```
async create(walletData: any, user: User): Promise<Wallet | APIError> {
    try {
        const wallet = await Wallet.create({...walletData, userId: user.id})
        return wallet.toJSON()
    } catch (e: any) {
        const errors = e.errors.map((error: any) => error.message)
        throw new APIError(errors)
    }
}

async increaseBalance(id: number, data: any): Promise<Wallet | APIError> {
    const {amount} = data
    const wallet = await Wallet.scope('nested').findByPk(id)
    if (wallet) {
        if (amount <= 0) {
            throw new APIError(`Got non-positive amount: ${amount}`)
        }
        console.log(amount)
        wallet.setDataValue('balance', wallet.balance + amount)
        await wallet.save()
        await wallet.reload()
        return wallet.toJSON()
    }
    throw new APIError('Wallet not found')
}
```

```
async decreaseBalance(id: number, data: any): Promise<Wallet | APIError> {
    const {amount} = data
    const wallet = await Wallet.scope('nested').findByPk(id)
    if (wallet) {
        if (amount <= 0) {
            throw new APIError(`Got non-positive amount: ${amount}`)
        }
        if (amount > wallet.getDataValue('balance')) {
            throw new APIError('Not enough money')
        }
        wallet.setDataValue('balance', wallet.balance - amount)
        await wallet.save()
        await wallet.reload()
        return wallet.toJSON()
    }
    throw new APIError('Wallet not found')
}
```

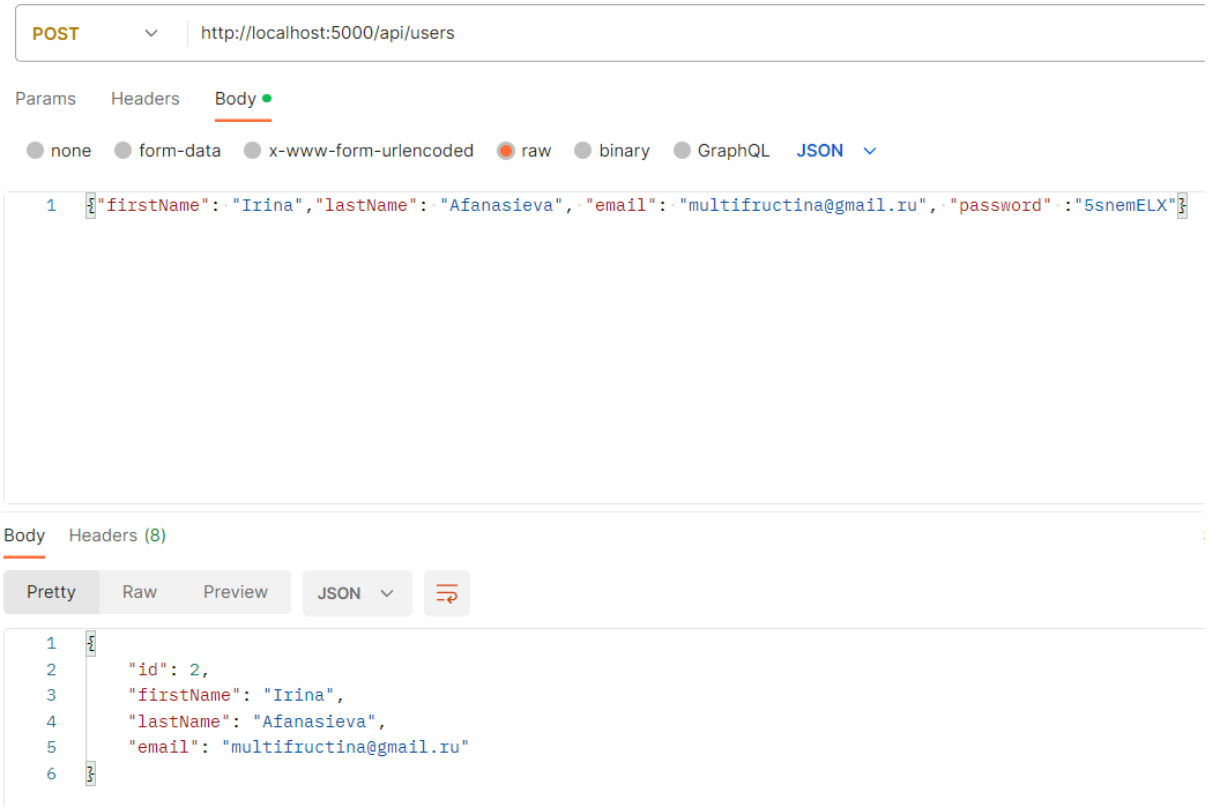
```
async buyCoin(id: number, data: any): Promise<Wallet | APIError> {
    const {amount, ticker} = data
    if (amount <= 0) {
        throw new APIError(`Got non-positive amount: ${amount}`)
    }
    const wallet = await Wallet.scope('nested').findByPk(id)
    const coin = await Coin.findByPk(ticker)
    if (wallet && coin) {
        let coinWallet = await CoinWallet.findOne({where: {coinId: ticker, walletId: id}})
        if (!coinWallet) {
            coinWallet = await CoinWallet.create({amount: 0, coinId: ticker, walletId: id})
        }
        let price: number
        try {
            price = await getCurrentPrice(ticker)
        } catch (e: any) {
            throw new APIError(e.message)
        }
        if (price * amount > wallet.getDataValue('balance')) {
            throw new APIError('Not enough money')
        }
        wallet.setDataValue('balance', wallet.getDataValue('balance') - price * amount)
        coinWallet.amount += amount
        await coinWallet.save()
        await wallet.save()
        await wallet.reload()
        return wallet.toJSON()
    } else if (!wallet) {
        throw new APIError('Wallet not found')
    } else {
        throw new APIError(`Unknown coin: ${ticker}`)
    }
}
```

```
async sellCoin(id: number, data: any): Promise<Wallet | APIError> {
    const {amount, ticker} = data
    if (amount <= 0) {
        throw new APIError(`Got non-positive amount: ${amount}`)
    }
    const wallet = await Wallet.scope('nested').findByPk(id)
    const coin = await Coin.findByPk(ticker)
    if (wallet && coin) {
        const coinWallet = await CoinWallet.findOne({where: {coinId: ticker, walletId: id}})
        if (coinWallet) {
            if (amount > coinWallet.amount) {
                throw new APIError(`You don't have ${amount} ${coin.name}`)
            }
            try {
                const price = await getCurrentPrice(ticker)
                wallet.setDataValue('balance', wallet.getDataValue('balance') + price * amount)
            } catch (e: any) {
                throw new APIError(e.message)
            }
            if (amount == coinWallet.amount) {
                await coinWallet.destroy()
            } else {
                coinWallet.amount -= amount
                await coinWallet.save()
            }
            await wallet.save()
            await wallet.reload()
            return wallet.toJSON()
        } else {
            throw new APIError(`You don't have any ${coin.name}`)
        }
    } else if (!wallet) {
        throw new APIError('Wallet not found')
    } else {
        throw new APIError(`Unknown coin: ${ticker}`)
    }
}
```

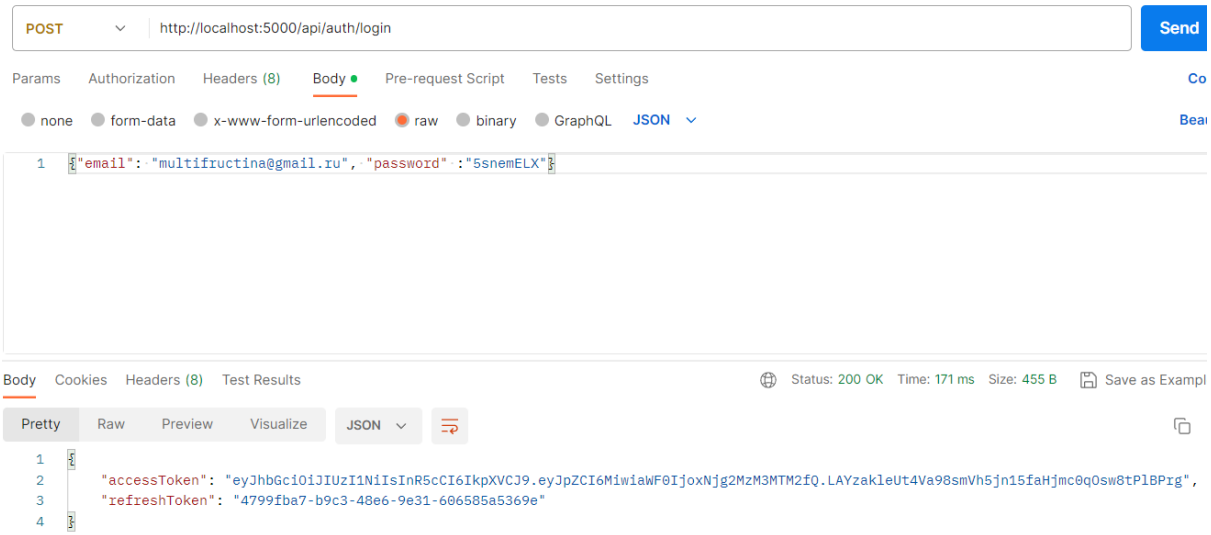
```
async sellAllCoin(id: number, data: any): Promise<Wallet | APIError> {
    const {ticker} = data
    const wallet = await Wallet.scope('nested').findByPk(id)
    const coin = await Coin.findByPk(ticker)
    if (wallet && coin) {
        const coinWallet = await CoinWallet.findOne({where: {coinId: ticker, walletId: id}})
        if (coinWallet) {
            const amount = coinWallet.amount
            try {
                const price = await getCurrentPrice(ticker)
                wallet.setDataValue('balance', wallet.getDataValue('balance') + price * amount)
            } catch (e: any) {
                throw new APIError(e.message)
            }
            await coinWallet.destroy()
            await wallet.save()
            await wallet.reload()
            return wallet.toJSON()
        } else {
            throw new APIError(`You don't have any ${coin.name}`)
        }
    } else if (!wallet) {
        throw new APIError('Wallet not found')
    } else {
        throw new APIError(`Unknown coin: ${ticker}`)
    }
}
```

Пример работы API:

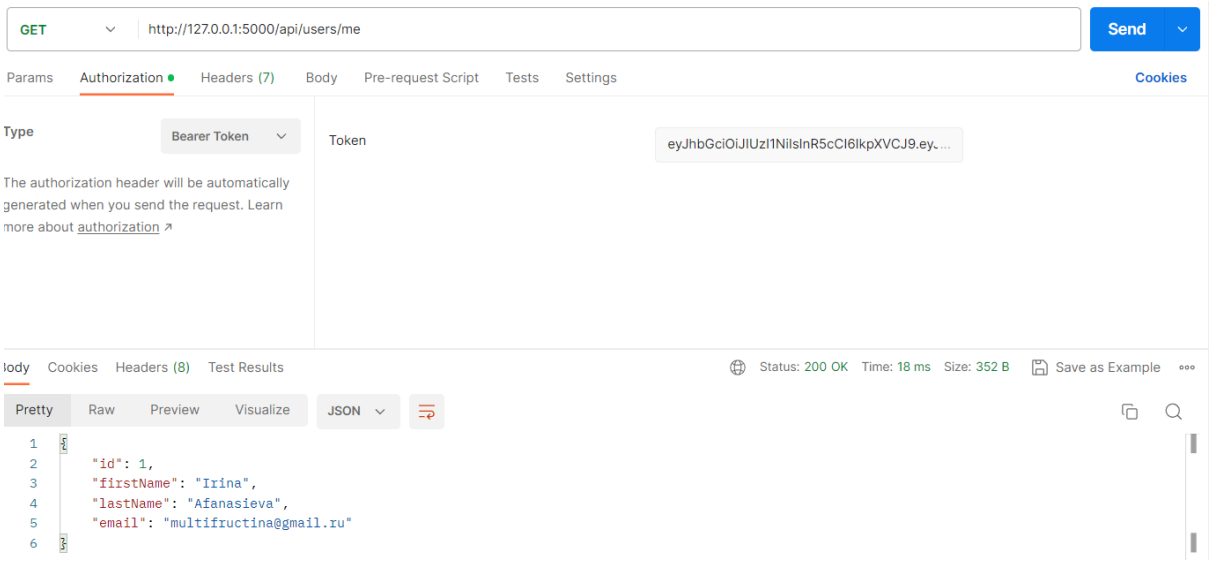
Создание пользователя:



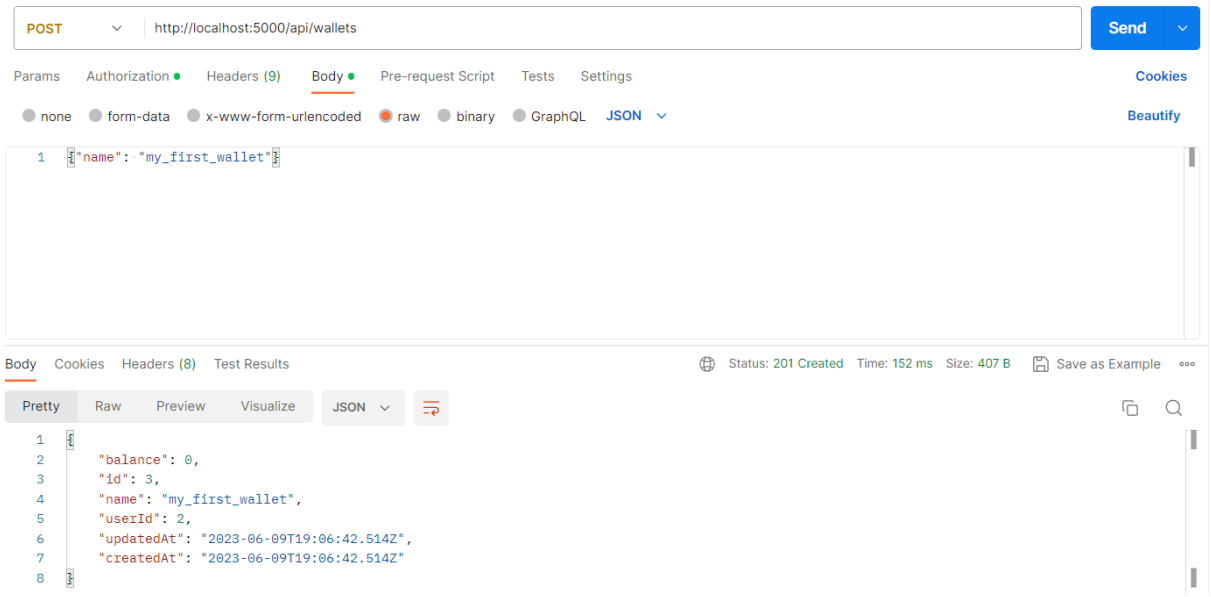
Вход:



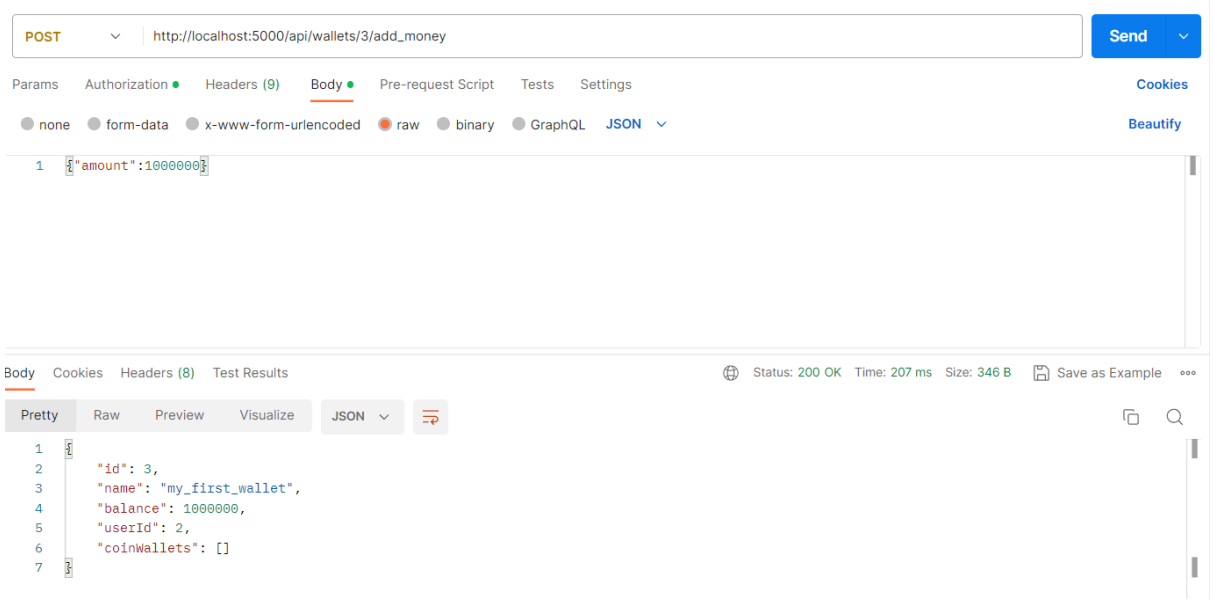
Получение профиля:



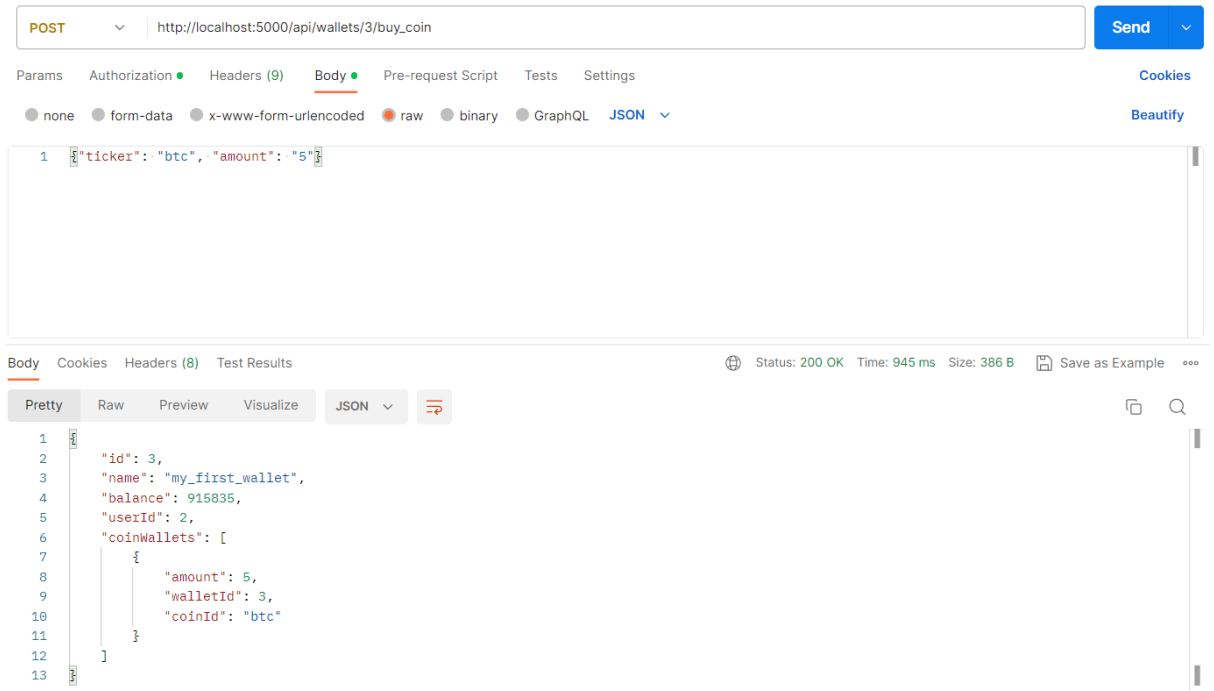
Создание портфеля:



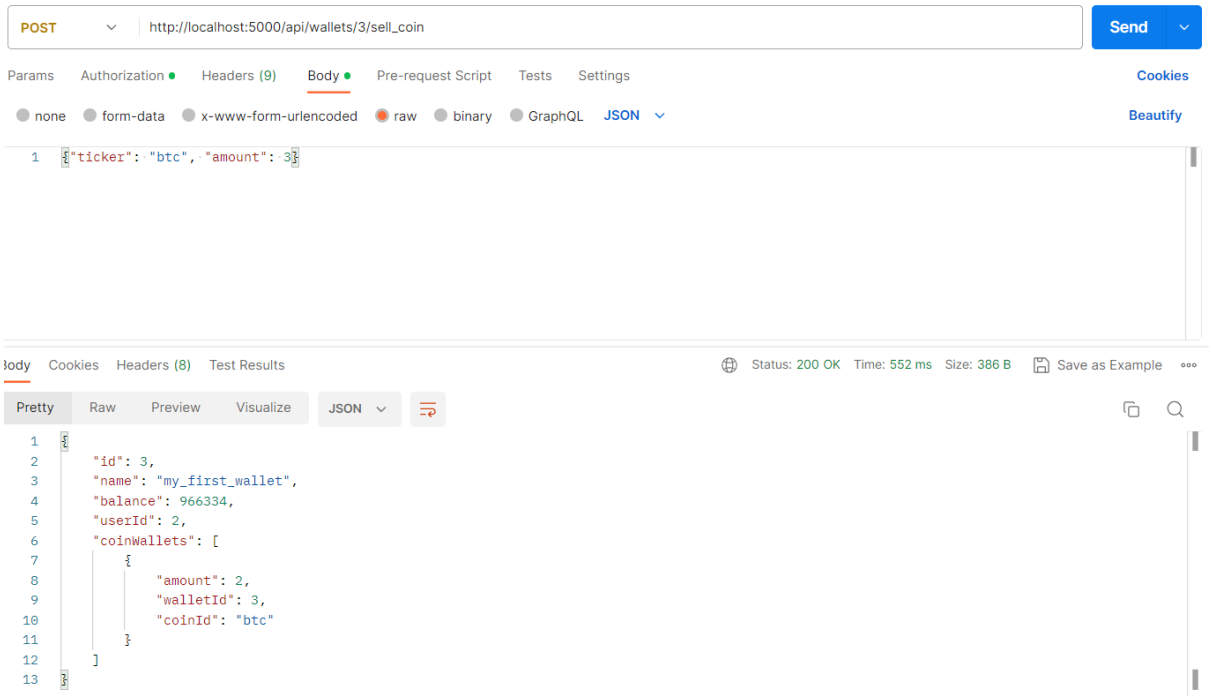
Добавление денег в портфель:



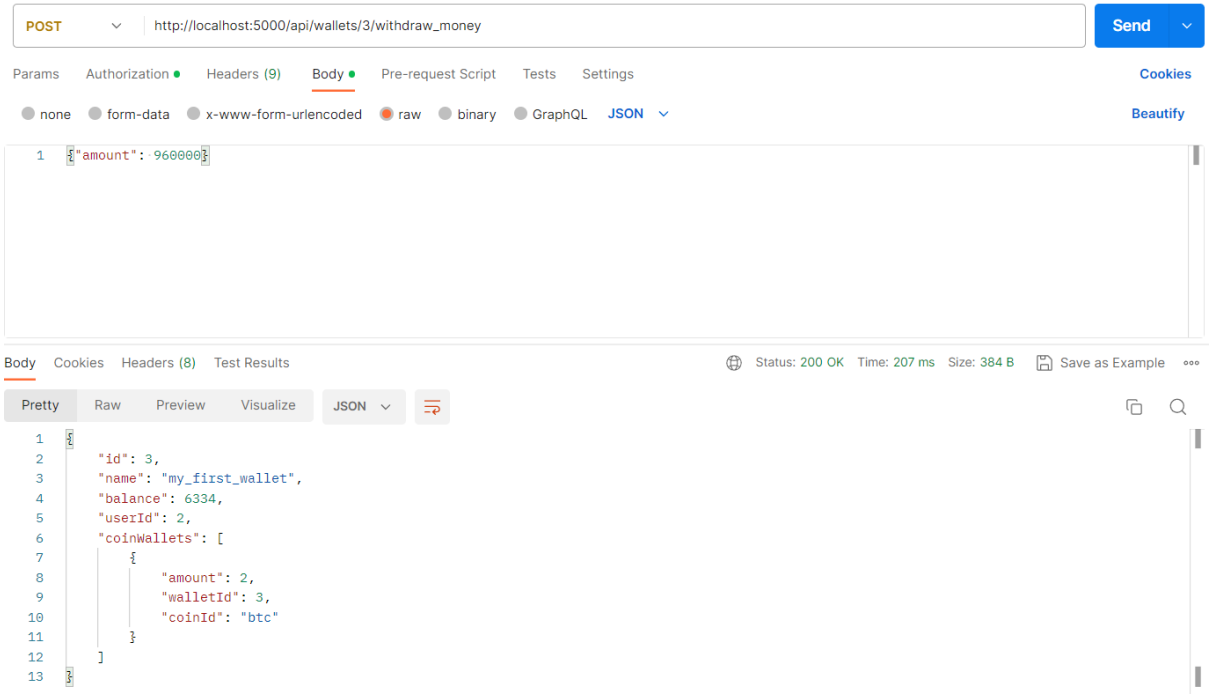
Покупка криптовалюты:



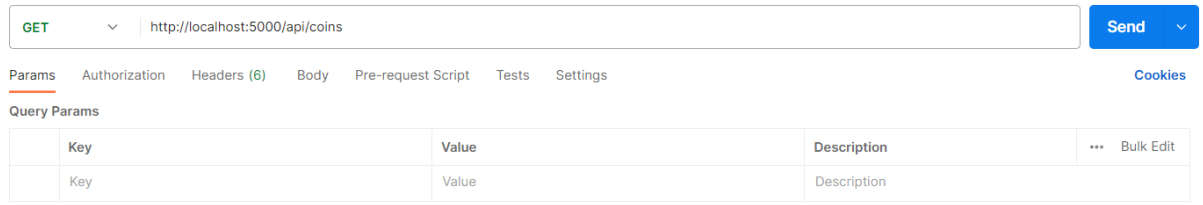
Продажа криптовалюты:

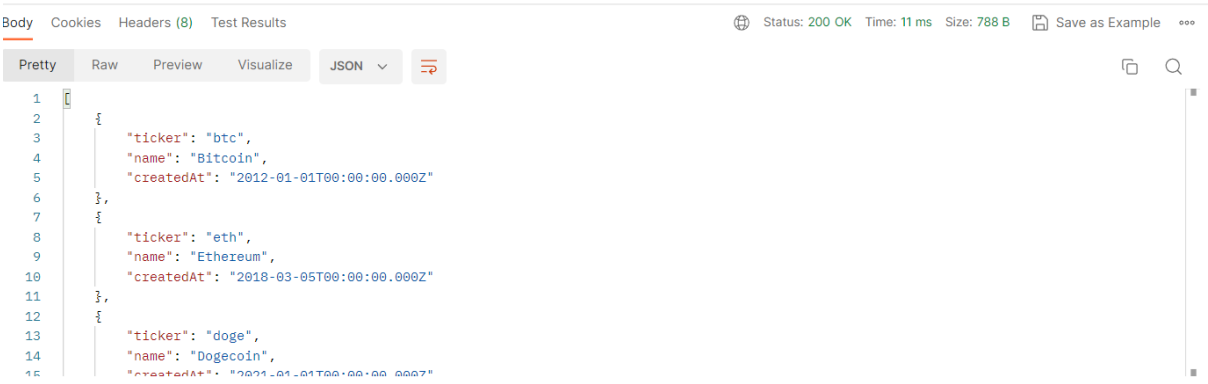


Вывод денег с биржи:

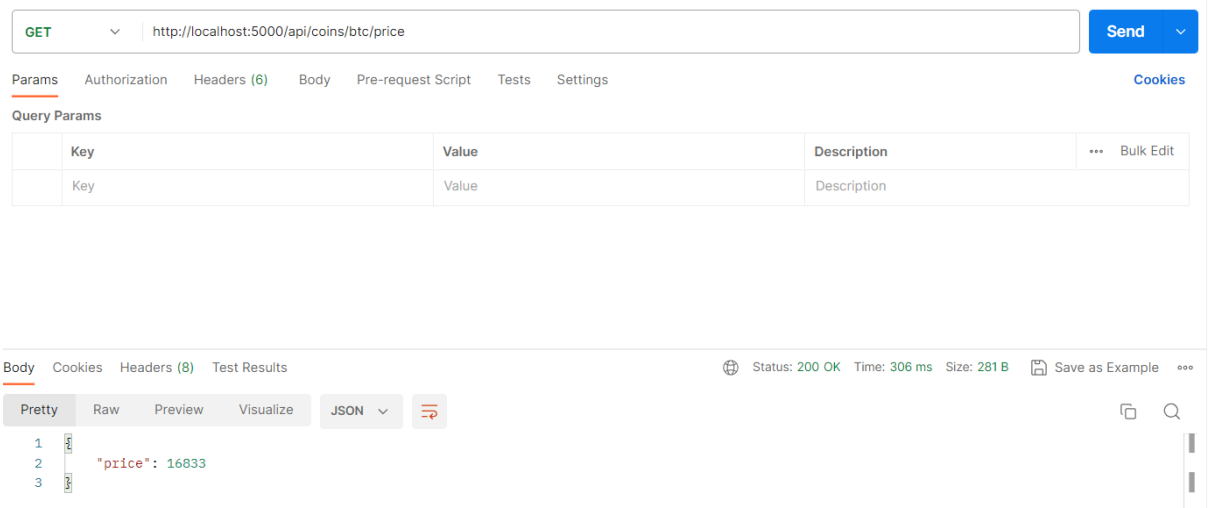


Список доступных криптовалют:





Цена криптовалюты в данный момент (используется API api.cryptowat.ch и данные биржи FTX):



Вывод

В ходе работы был реализован RESTful API для сайта криптобиржи с использованием фреймворка Express, ORM Sequelize, Axios и другими библиотеками.