

Solidity burza v decentralizovanom svete

(Zadanie 2)

LS 2024/2025

Contents

Odpovede na otázky	3
Riešenie doimplementácie.....	4
token.sol:.....	4
exchange.sol:.....	6
exchange.js:.....	11
Bonus -> Pridanie javascript funkcionality na zobrazenie stavu účtu:	13
exchange.js:.....	13
index.html:	14
exchange.sol:.....	14
Bonus -> Pridanie testov:	15
Spustenie testov:.....	15
Záver:.....	15

Odpovede na otázky

Otázka1: Vysvetlite, prečo pridávanie a odoberanie likvidity na vašej burze nezmení výmenný kurz.

Odpoveď: Pri každom pridaní ETH (wei) do LP (LiquidityPool) sa na smart kontrake vypočíta primeraný počet pridaných tokenov podľa aktuálneho výmenného kurzu.

`amountTokens = (token.balanceOf(address(this)) * msg.value) / address(this).balance;`

Výmenný kurz tak zostáva stabilný, pretože pomer medzi ETH a tokenmi v poole sa nemení.

Otázka2: K bonusu - Vysvetlite svoju schému odmeňovania poskytovateľov likvidity a zdôvodnite rozhodnutia o dizajne, ktoré ste urobili. Ako splňa požiadavky na odmeny za likviditu uvedené v sekciu 7?

Odpoveď: Pri každej výmene (swape) zostáva malá časť tokenov alebo wei priamo v LP. Táto časť predstavuje poplatok za vykonanie výmeny, ktorý je v našej implementácii nastavený na 3 % z celkovej sumy swapu. Namiesto toho, aby sa celá suma vrátila používateľovi, 3 % zostávajú uložené v smart kontrakte. Vďaka tomu sa pri každom swape zvyšuje celková hodnota likviditného pulzu, čo znamená, že rastie aj hodnota podielov poskytovateľov likvidity. Keďže tieto podiel evidujeme ako percentá, nie je potrebné ich manuálne aktualizovať. Pri každej výmene sa hodnota tokenov a wei automaticky prerozdelí podľa existujúcich podielov jednotlivých poskytovateľov.

Otázka3: Popište aspoň jednu metódu, ktorú ste použili na minimalizáciu spotreby gas pri kontrakte burzy. Prečo bola táto metóda efektívna?

Odpoveď: Snažili sme sa optimalizovať kód tým, že sme všade, kde to bolo možné, použili kľúčové slovo constant, čím sme výrazne znížili náklady na čítanie hodnôt (čítanie z constant premenných stojí len 3 gas oproti približne 2100 gas pri čítaní zo storage). Zároveň sme opakujúce sa výpočty vykonali len raz, ich výsledky sme uložili do lokálnych premenných a tie sme ďalej používali v kóde. Týmto prístupom sme sa vyhli zbytočnému opakovaniu operácií (napr. násobenie stojí približne 5 gas), čo pri zložitejších vzorcoch predstavuje výraznú úsporu. Celkovo sme tak znížili spotrebú gasu a optimalizovali náklady na vykonanie transakcie.

Otázka4a: Koľko času ste strávili na zadani? (+30h)

Otázka4b: Aká je jedna vec, ktorá by bola užitočná, keby ste ju vedeli predtým, ako ste začali pracovať na zadani?

Viac o manipulácií a reprezentácií reálnych čísel v prostredí, kde existujú jedine INT (zaokrúhľovanie, prevencia overflowu a underflowu).

Otázka4c: Keby ste mohli zmeniť jednu vec v tomto zadani, čo by to bolo? (Nič)

Otázka4d: Prosím pridajte nám akýkolvek feedback alebo spätnú väzbu, ktorý máte na mysli alebo na srdci 😊. (Dobré zadanie, ľúbilo sa nám)

Riešenie doimplementácie

token.sol:

```
contract Token is Ownable, ERC20
{
    string private constant _symbol = 'TT';                                // TODO: G
    string private constant _name = 'TokenToken';                            //

    bool mint_enabled = true;

    uint private _totalSupply;
    mapping(address => uint) private _balances;
    mapping(address => mapping(address => uint)) private _allowances;

    constructor() ERC20(_name, _symbol) {}
}
```

Náš token je navrhnutý tak, aby vychádzal zo štandardu ERC20 a implementoval jeho základné funkcionality. Obsahuje vlastné meno a symbol. Premenná `_totalSupply` uchováva aktuálny počet všetkých doteraz emitovaných (mintovaných) tokenov. Na sledovanie zostatku tokenov pre jednotlivé účty používame dátovú štruktúru mapping, ktorá funguje ako efektívnejšie pole s použitím hash funkcie – ku každej adrese (účtu) priradujeme počet jej tokenov. Štruktúra `_allowance` funguje podobne, ale ako dvojrozmerné mapovanie, kde ukladáme informáciu o tom, koľko tokenov jeden účet povolil miňať inému účtu (`_allowance[owner][spender]`).

```
function transfer(address to, uint256 amount) public virtual override returns (bool)
{
    require(to != address(0), "Transfer to zero address");
    require(_balances[msg.sender] >= amount, "insufficient balance");

    _balances[msg.sender] -= amount;
    _balances[to] += amount;

    emit Transfer(msg.sender, to, amount);
    return true;
}
```

Základná funkcia na posielanie tokenov vychádza zo štandardu ERC20 – metóda `transfer`. Táto funkcia umožňuje, aby odosielateľ (volajúci funkciu) poslal požadovaný počet tokenov na adresu prijímateľa. Pri tomto procese sa z účtu odosielateľa odpočítava príslušný počet tokenov a tieto tokeny sa následne pripisujú na účet prijímateľa. Ak je prevod vykonaný správne, informácie o odosielateľovi, prijímateľovi a počte prevedených tokenov sa zaznamenajú na blockchain.

```
function allowance(address owner, address spender) public view virtual override returns (uint256)
{
    require(owner != address(0), "Transfer to zero address");
    require(spender != address(0), "Transfer from zero address");

    return _allowances[owner][spender];
}

function approve(address spender, uint256 amount) public virtual override returns (bool)
{
    require(spender != address(0), "Approve to zero address");

    _allowances[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

function transferFrom(address from, address to, uint256 amount) public virtual override returns (bool)
{
    require(from != address(0), "From zero address");
    require(to != address(0), "To zero address");
    uint currentAllowance = _allowances[from][msg.sender];
    require(currentAllowance >= amount, "Insufficient allowance");
    require(_balances[from] >= amount, "Insufficient balance");

    _balances[from] -= amount;
    _balances[to] += amount;

    _allowances[from][msg.sender] -= amount;
    emit Transfer(from, to, amount);
    return true;
}
```

Druhým spôsobom prevodu tokenov podľa ERC20 je funkcia transferFrom. Tá umožňuje, aby niekto iný mohol previesť naše tokeny, ak mu na to dáme povolenie pomocou funkcie approve. Toto povolenie (allowance) určuje, koľko tokenov môže daný používateľ minút z nášho účtu. Keď je transferFrom úspešne vykonaný, tokeny sa odpočítajú z nášho účtu, pripíšu sa príjemcovi a zároveň sa zníži allowance o prevedenú sumu. Celá transakcia sa zaznamená na blockchain.

```
// =====
//          FUNCTIONS TO IMPLEMENT
// =====

// Function _mint: Create more of your tokens.
// You can change the inputs, or the scope of your function, as needed.
// Do not remove the AdminOnly modifier!
function mint(uint amount) public onlyOwner
{
    require(mint_enabled, "Mint has already been disabled.");
    _totalSupply += amount;
    _balances[msg.sender] += amount;
    emit Transfer(address(0), msg.sender, amount);
}

// Function _disable_mint: Disable future minting of your token.
// You can change the inputs, or the scope of your function, as needed.
// Do not remove the AdminOnly modifier!
function disable_mint() public onlyOwner
{
    require(mint_enabled, "Mint has already been disabled.");
    mint_enabled = false;
}
```

Na vytváranie nových tokenov sme implementovali funkciu mint, ktorá je jediným spôsobom, ako emitovať nové tokeny. Túto funkciu môže volať len vlastník kontraktu (onlyOwner) a všetky nové tokeny sa priradujú na jeho účet. Po deaktivovaní mintovania (nastavením prepínača) už nie je možné vytvárať ďalšie tokeny – funkcia mint sa tým trvalo znefunkční.

exchange.sol:

```
event LiquidityAdded(
    address indexed provider,
    uint256 tokenAmount,
    uint256 ethAmount,
    uint256 liquidityMinted
);

event LiquidityRemoved(
    address indexed provider,
    uint256 tokenAmount,
    uint256 ethAmount,
    uint256 liquidityMinted
);
```

Na začiatku súboru sú definované eventy. Event LiquidityAdded hovorí o pridaní liquidity do celkového poolu, a LiquidityRemoved hovorí o odstránení liquidity z poolu.

```
string public constant exchange_name = 'CeilingStreet';

address constant tokenAddr = 0x9D8946A8A0c5a583Bf05bE83B981406182ac9d2f;           // TODO: paste token contract address here
Token public token = Token(tokenAddr);

// Liquidity pool for the exchange
uint private token_reserves = 0;
uint private eth_reserves = 0;

mapping(address => uint) private lps;

// Needed for looping through the keys of the lps mapping
address[] private lp_providers;

// liquidity rewards
uint private constant swap_fee_numerator = 3;
uint private constant swap_fee_denominator = 100;

uint private constant rate_decimals = 3;

// Constant: x * y = k
uint private k;
```

Ako premenné vo vnútri kontraktu si uchovávame:

názov kontraktu (exchange_name), token, ktorý budeme vymieňať za ethery a naopak, aj s jeho adresou (tokenAddr, token), celkový počet tokenov a etherov v pooli (token_reserves, eth_reserves), mapovanie adries liquidity providerov na percentuálnu čiastku celkového poolu (lps), ktorú vlastnia, pole providerov lp_providers (na umožnenie cyklovania cez mapu lps),

swap_fee_numerator a swap_fee_denominator na vytvorenie zlomku, ktorý predstavuje veľkosť dane za vymieňanie etherov za tokeny a naopak, rate_decimals, ktorý definuje koľko desatiných miest má percentuálny podiel jednotlivých providerov (v našej implementácii tri), a konšanta k na výpočet prevodu medzi ethermi a tokenmi.

```
// Function createPool: Initializes a liquidity pool between your Token and ETH.  
// ETH will be sent to pool in this transaction as msg.value  
// amountTokens specifies the amount of tokens to transfer from the liquidity provider.  
// Sets up the initial exchange rate for the pool by setting amount of token and amount of ETH.  
function createPool(uint amountTokens) external payable onlyOwner  
{  
    // This function is already implemented for you; no changes needed.  
  
    // require pool does not yet exist:  
    require (token_reserves == 0, "Token reserves was not 0.");  
    require (eth_reserves == 0, "ETH reserves was not 0.");  
  
    // require nonzero values were sent  
    require (msg.value > 0, "Need eth to create pool.");  
    uint tokenSupply = token.balanceOf(msg.sender);  
    require (amountTokens <= tokenSupply, "Not have enough tokens to create the pool.");  
    require (amountTokens > 0, "Need tokens to create pool.");  
  
    bool success = token.transferFrom(msg.sender, address(this), amountTokens);  
    require(success, "Unsuccessful transfer.");  
  
    token_reserves = token.balanceOf(address(this));  
    eth_reserves = msg.value;  
    k = token_reserves * eth_reserves;  
}
```

Funkcia createPool slúži na inicializáciu kontraktu. Dá sa zavolať len za predpokladu, že liquidity pool je prázdny. Pridá všetky poslané ethery pri volaní funkcie do poolu, ako aj množstvo tokenov, ktoré bolo špecifikované v argumente. Tokeny posielala na svoju adresu z adresy voľača funkcie, a preto je dôležité, aby mal kontrakt pred zavolaním createPool nastavený dostatočný allowance. Nakoniecs a vypočíta nová hodnota k.

```
// Function addLiquidity: Adds liquidity given a supply of ETH (sent to the contract as msg.value).
// You can change the inputs, or the scope of your function, as needed.
// exchange_rate => (token / ether) 10 ^ 8
function addLiquidity(uint max_exchange_rate, uint min_exchange_rate) external payable
{
    uint tokenBalanceCurr = token.balanceOf(address(this));
    uint weiBalanceCurr = address(this).balance;
    uint weiBalanceAfter = weiBalanceCurr - msg.value;
    //***** TODO: Implement this function *****/
    // require nonzero values were sent
    require (msg.value > 0, "Need eth to add liquidity.");
    uint exchange_rate = (tokenBalanceCurr * (10 ** rate_decimals)) / weiBalanceAfter;
    require (exchange_rate <= max_exchange_rate, "Exchange rate is larger than the max exchange rate.");
    require (exchange_rate >= min_exchange_rate, "Exchange rate is smaller than the min exchange rate.");
    uint tokenSupply = token.balanceOf(msg.sender);
    uint amountTokens = (tokenBalanceCurr * msg.value) / weiBalanceAfter;

    require (amountTokens <= tokenSupply, "Not have enough tokens to add to liquidity.");

    token.transferFrom(msg.sender, address(this), amountTokens);

    uint previous_liquidity = lps[msg.sender];
    if (lps[msg.sender] == 0)
        lp_providers.push(msg.sender);

    token_reserves = tokenBalanceCurr + amountTokens;
    eth_reserves = weiBalanceCurr;
    k = token_reserves * eth_reserves;

    for (uint i = 0; i < lp_providers.length; i++) {
        uint lp_balance = lps[lp_providers[i]] * weiBalanceAfter;
        if (lp_providers[i] == msg.sender)
            lp_balance += msg.value * (10 ** rate_decimals);
        lps[lp_providers[i]] = lp_balance / eth_reserves;
    }

    emit LiquidityAdded(msg.sender, amountTokens, msg.value, lps[msg.sender] - previous_liquidity);
}
```

Funkcia na začiatku vykoná viac krát sa opakujúce výpočty, aby sme šetrili gas. Následne sa vypočítava aktuálny exchange rate, a skontroluje sa, či sa nachádza v stanovených hraniciach. Následne sa vypočítava množstvo tokenov, ktoré sa musí poslať na kontrakt, aby bol zachovaný správny pomer etherov a tokenov v poole. Tieto tokeny sa pošlú od msg.sender na adresu kontraktu, kvôli čomu je dôležité, aby msg.sender pred volaním tejto funkcie stanovil dostatočný allowance kontraktu.

Skontroluje sa aj, či sa msg.sender nachádza v poli liquidátorov. Ak sa tam nenachádza, tak ho do poľa pridáme. Aktualizujeme premenné počtu tokenov a eth v kontrakte, a vypočítame nové k.

Nakoniec vypočítame pre každého likvidátora novú hodnotu podielu. Výpočet sa vykonáva v cykle, kde vypočítame veľkosť starého nároku každého likvidátora na ethery, a podľa tohto nároku a nového počtu etherov vypočítame nový podiel. Pre msg.sender pridáme ku nároku aj novo poslané ethery.

Event pridania likvidity oznamíme použitím emit.

```
// Function removeLiquidity: Removes liquidity given the desired amount of ETH to remove.
// You can change the inputs, or the scope of your function, as needed.
function removeLiquidity(uint amountETH, uint max_exchange_rate, uint min_exchange_rate) public
{
    uint tokenBalanceCurr = token.balanceOf(address(this));
    uint weiBalanceCurr = address(this).balance;
    //***** TODO: Implement this function *****/
    uint exchange_rate = (tokenBalanceCurr * (10 ** rate_decimals)) / weiBalanceCurr;
    require (exchange_rate <= max_exchange_rate, "Exchange rate is larger than the max exchange rate.");
    require (exchange_rate >= min_exchange_rate, "Exchange rate is smaller than the min exchange rate.");
    uint lp_balance = (lps[msg.sender] * weiBalanceCurr) / (10 ** rate_decimals);
    require (lp_balance >= amountETH, "Not enough liquidity in pool for this provider.");

    uint amountTokens = (tokenBalanceCurr * amountETH) / weiBalanceCurr;

    require(eth_reserves - amountETH > 0, "Not enough ethers to transfer.");
    require(token_reserves - amountTokens > 0, "Not enough tokens to transfer.");

    uint previous_liquidity = lps[msg.sender];

    token_reserves = tokenBalanceCurr - amountTokens;
    eth_reserves = weiBalanceCurr - amountETH;
    k = token_reserves * eth_reserves;

    uint lp_index;
    for (uint i = 0; i < lp_providers.length; i++) {
        lp_balance = lps[lp_providers[i]] * weiBalanceCurr;
        if (lp_providers[i] == msg.sender) {
            lp_balance -= amountETH * (10 ** rate_decimals);
            lp_index = i;
        }
        lps[lp_providers[i]] = lp_balance / eth_reserves;
    }

    if (lps[msg.sender] == 0)
        removeLP(lp_index);

    (bool success, ) = payable(msg.sender).call{value: amountETH}("");
    require(success, "ETH transfer failed");
    token.transfer(msg.sender, amountTokens);

    emit LiquidityRemoved(msg.sender, amountTokens, amountETH, previous_liquidity - lps[msg.sender]);
}
```

Funguje podobne ako addLiquidity(). Tiež skontrolujeme, či sa exchange rate nachádza v intervale stanovenom argumentami. Na rozdiel od addLiquidity() musíme skontrolovať, či msg.sender má dostatočné množstvo likvidity v poole. To overíme kontrolou počtu tokenov a etherov, na ktoré má daná adresa nárok. Podobne ako pri addLiquidity(), aj tu prepočítame percentuálne nároky jednotlivých likvidátorov. Naroziel od addLiquidity() počet etherov odrátame od nároku msg.sender adresy, a prípadne túto adresu odstránime z pola likvidátorov v prípade, že podiel daného likvidátora na celkový pool klesla na nulu.

RemoveAllLiquidity() funguje rovnako ako removeLiquidity(), až na to, že namiesto posielania množstva odobratého etherea z poolu sa dané množstvo vypočíta podľa celkového podielu voľača funkcie na pool. Odoberie sa teda všetka likvidita pre danú adresu.

```
function getRateDecimals() public pure returns (uint) {
    return rate_decimals;
}

function getProviderLiquidity(address account) public view returns (uint) {
    return (lps[account] * address(this).balance) / (10 ** rate_decimals);
}
```

Zapuzdrovacie funkcie na získanie počtu desatinných miest pre reprezentáciu podielu poolu jednotlivých likvidátorov a získanie likvidity špecifického providera. GetProviderLiquidity vracia podiel v jednotkách wei (teda počet wei, na ktoré má daný likvidátor nárok).

```
// Function swapTokensForETH: Swaps your token with ETH
// You can change the inputs, or the scope of your function, as needed.
function swapTokensForETH(uint amountTokens, uint max_exchange_rate) external payable
{
    //***** TODO: Implement this function *****/
    uint exchange_rate = (token.balanceOf(address(this)) * (10 ** rate_decimals)) / (address(this).balance - msg.value);
    require (exchange_rate <= max_exchange_rate, "Exchange rate is larger than the max exchange rate.");

    uint amountETH = eth_reserves - (k / (token_reserves + amountTokens));
    uint amountETH_transferred = (amountETH * (swap_fee_denominator - swap_fee_numerator)) / swap_fee_denominator;

    require(eth_reserves - amountETH_transferred > 0, "Not enough ethers to transfer.");

    token.transferFrom(msg.sender, address(this), amountTokens);

    (bool success, ) = payable(msg.sender).call{value: amountETH_transferred}("");
    require(success, "ETH transfer failed");

    token_reserves = token_reserves + amountTokens;
    eth_reserves = eth_reserves - amountETH_transferred;
}

// Function swapETHForTokens: Swaps ETH for your tokens
// ETH is sent to contract as msg.value
// You can change the inputs, or the scope of your function, as needed.
function swapETHForTokens(uint max_exchange_rate) external payable
{
    //***** TODO: Implement this function *****/
    uint exchange_rate = ((address(this).balance - msg.value) * (10 ** rate_decimals)) / token.balanceOf(address(this));
    require (exchange_rate <= max_exchange_rate, "Exchange rate is larger than the max exchange rate.");

    uint amountTokens = token_reserves - (k / (eth_reserves + msg.value));
    uint amountTokens_transferred = (amountTokens * (swap_fee_denominator - swap_fee_numerator)) / swap_fee_denominator;

    require(token_reserves - amountTokens_transferred > 0, "Not enough tokens to transfer.");

    token.transfer(msg.sender, amountTokens_transferred);

    token_reserves = token_reserves - amountTokens_transferred;
    eth_reserves = eth_reserves + msg.value;
}
```

Swapovacie funkcie slúžia na konvertovanie tokenov na wei alebo naopak. Najprv skontrolujú, či sa aktuálny exchange rate nachádza v stanovenom intervale. Potom podľa vzorca pre k vypočítajú adekvátne množstvo wei alebo tokenov, ktoré sa má poslať voľačovi funkcie.

Následne sa toto množstvo zmenší podľa veľkosti swap_fee. Vďaka tomuto zmenšovaniu odosielaného množstva tokenov a wei sa pasívne odmeňujú likvidátori. Pretože ich percentuálny nárok na celkovú veľkosť poolu sa nemení, zväčšovaním veľkosti celkového poolu zväčšujeme aj množstvo wei a tokenov, na ktoré majú nárok. Pri funkcií swapTokensForETH je dôležité, aby sme pred volaním funkcie dali kontraktu dostatočný allowance, pretože táto funkcia sa pokúsi poslať si množstvo tokenov na svoju adresu z adresy msg.sender.

Je dôležité podotknúť aj to, že ako argument do funkcie neposielame min_exchange_rate, pretože nechceme volanie funkcie prerušiť pri malých množstvách exchange rate. Malá hodnota je totiž pre nás výhodná, keďže nám dovolí vymeniť wei alebo tokeny za väčšie množstvo tokenov alebo wei.

exchange.js:

```
async function getAccountBalance(account) {
    let token_balance = BigInt(await token_contract.connect(provider.getSigner(account)).balanceOf(account));
    let eth_balance = BigInt(await provider.getBalance(account));
    let liquidity = BigInt(await exchange_contract.connect(provider.getSigner(account)).getProviderLiquidity(account));
    return {
        tokens: token_balance,
        eth: eth_balance,
        liq: liquidity
    }
}
```

Funkcia getAccountBalance() vráti počet tokenov a wei na danom účte. Taktiež sa pozrie na exchange.sol kontrakt na blockchaine, a vráti nárok (vo wei) daného účtu z celkového poolu na kontrakte.

```
/** ADD LIQUIDITY */
async function addLiquidity(amountEth, maxSlippagePct) {
    /* TODO: ADD YOUR CODE HERE */
    let state = await getPoolState();
    let rateDecimals = await getRateDecimals();
    let max_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) + maxSlippagePct * (10 ** (rateDecimals - 2)));
    let min_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) - maxSlippagePct * (10 ** (rateDecimals - 2)));

    let amount_tokens = Math.floor((state.token_liquidity * amountEth) / state.eth_liquidity);
    await token_contract.connect(provider.getSigner(defaultAccount)).approve(exchange_address, amount_tokens);
    await exchange_contract.connect(provider.getSigner(defaultAccount)).addLiquidity(max_exchange_rate, min_exchange_rate,
        { value: ethers.utils.parseUnits(amountEth.toString(), "wei") });
}
```

Zavolá funkciu pre pridanie likvidity na danom smart kontrakte. Na začiatku potrebuje získať aktuálny pomer tokenov a wei, aby sme dokázali určiť maximálnu a minimálnu odchylku. Tieto odchylky vypočítame s dohľadom na správny počet desatinných miest vo finálnom integeri.

Počet desatinných miest získame z premennej rateDecimals, ktorú inicializujeme zavolením getRatedDecimals, čo je funkcia volajúca zapuzdrovaciu funkciu na smart kontrakte. Vďaka tomuto mechanizmu máme správny počet desatinných miest aj v prípade dynamického počtu desatinných miest na smart kontrakte (pretože celý mechanizmus závisí jedine od vrátenej hodnoty z getRateDecimals()).

Pred zavolením samotnej funkcie pre zväčšenie likvidity musíme zavolať funkciu approve, aby exchange smart kontrakt mohol presunúť špecifikovaný počet tokenov.

```
/** REMOVE LIQUIDITY */
async function removeLiquidity(amountEth, maxSlippagePct) {
    /* TODO: ADD YOUR CODE HERE */
    let state = await getPoolState();
    let rateDecimals = await getRateDecimals();
    let max_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) + maxSlippagePct * (10 ** (rateDecimals - 2)));
    let min_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) - maxSlippagePct * (10 ** (rateDecimals - 2)));

    await exchange_contract.connect(provider.getSigner(defaultAccount)).removeLiquidity(amountEth, max_exchange_rate, min_exchange_rate);
}

async function removeAllLiquidity(maxSlippagePct) {
    /* TODO: ADD YOUR CODE HERE */
    let state = await getPoolState();
    let rateDecimals = await getRateDecimals();
    let max_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) + maxSlippagePct * (10 ** (rateDecimals - 2)));
    let min_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) - maxSlippagePct * (10 ** (rateDecimals - 2)));

    await exchange_contract.connect(provider.getSigner(defaultAccount)).removeAllLiquidity(max_exchange_rate, min_exchange_rate);
}
```

Funkcie `removeLiquidity()` a `removeAllLiquidity()` fungujú veľmi podobne. Obe funkcie slúžia na odstraňovanie likvidity z daného účtu, s maximálnou a minimálnou odchyľkou exchange rate. Na rozdiel od `addLiquidity()`, nemusíme pridať exchange kontraktu pred volaním funkcie žiadny allowance.

```
async function swapTokensForETH(amountToken, maxSlippagePct) {
    /* TODO: ADD YOUR CODE HERE */
    let state = await getPoolState();
    let rateDecimals = await getRateDecimals();
    let max_exchange_rate = Math.floor(state.token_eth_rate * (10 ** rateDecimals) + maxSlippagePct * (10 ** (rateDecimals - 2)));

    await token_contract.connect(provider.getSigner(defaultAccount)).approve(exchange_address, amountToken);
    await exchange_contract.connect(provider.getSigner(defaultAccount)).swapTokensForETH(amountToken, max_exchange_rate);
}

async function swapETHForTokens(amountEth, maxSlippagePct) {
    /* TODO: ADD YOUR CODE HERE */
    let state = await getPoolState();
    let rateDecimals = await getRateDecimals();
    let max_exchange_rate = Math.floor(state.eth_token_rate * (10 ** rateDecimals) + maxSlippagePct * (10 ** (rateDecimals - 2)));
    await exchange_contract.connect(provider.getSigner(defaultAccount)).swapETHForTokens(max_exchange_rate,
        { value: ethers.utils.parseUnits(amountEth.toString(), "wei") });
}

async function getRateDecimals() {
    return await exchange_contract.connect(provider.getSigner(defaultAccount)).getRateDecimals();
}
```

Výmena tokenov za wei a naopak znova potrebuje vypočítať odchyľku. Na rozdiel od `addLiquidity()` a `removeLiquidity()` nám stačí horná hranica odchyľky (z dôvodov vysvetlených vyššie). Pri výmene tokenov za wei treba určiť pred zavolaním funkcie `swapTokensForETH` aj allowance.

Bonus -> Pridanie javascript funkcionality na zobrazenie stavu účtu:



exchange.js:

```
init().then(() => {
  // fill in UI with current exchange rate:
  account = $("#myaccount").val();
  getAccountBalance(account).then((account_balance) => {
    getPoolState().then((poolState) => {
      $("#eth-token-rate-display").html("1 ETH = " + poolState['token_eth_rate'] + " " + token_symbol);
      $("#token-eth-rate-display").html("1 " + token_symbol + " = " + poolState['eth_token_rate'] + " ETH");

      $('#account-token-display').html("Token amount: " + account_balance.tokens.toLocaleString());
      $('#account-eth-display').html("ETH amount: " + (account_balance.eth).toLocaleString());
      $('#account-liquidity-display').html("Liquidity: " + account_balance.liq);
      console.log("Liquidity: " + account_balance.liq);

      $("#token-reserves").html(poolState['token_liquidity'] + " " + token_symbol);
      $("#eth-reserves").html(poolState['eth_liquidity'] + " ETH");
    });
  });
});
```

Funkcia init bola modifikovaná na jednoduché zobrazenie množstva tokenov, wei a likvidity na účte aktuálne vybranom v select elemente s id #myaccount.

```
$("#myaccount").change(function() {
    account = $(this).val();
    getAccountBalance(account).then((account_balance) => {
        $('#account-token-display').html("Token amount: " + account_balance.tokens.toLocaleString());
        $('#account-eth-display').html("ETH amount: " + (account_balance.eth).toLocaleString());
        $('#account-liquidity-display').html("Liquidity: " + account_balance.liq);
        console.log("Liquidity: " + account_balance.liq);
    });
});
```

Jednoduché zobrazenie množstva tokenov, wei a likvidity na danom účte po každom zmenení hodnoty v select elemente.

index.html:

```
<div class="panel panel_right">
    <h2>My Account</h2>
    <select id="myaccount" class="account"></select>
    <div id="account-eth-display"></div>
    <div id="account-token-display"></div>
    <div id="account-liquidity-display"></div>
</div>
```

Pridanie kontajnerov pre zobrazenie informácií o účte.

exchange.sol:

```
function getProviderLiquidity(address account) public view returns (uint) {
    return (lps[account] * address(this).balance) / (10 ** rate_decimals);
}
```

Funkcia vracia podiel daného účtu na celkovom množstve poolu v jednotkách wei (teda počet wei, na ktoré má daný likvidátor nárok).

Bonus -> Pridanie testov:

Vytvorili sme testy na pokrytie všetkých funkcií vo vnútri token.sol, a troch funkcií vo vnútri exchange.sol. Pokryté funkcie v exchange.sol sú špecificky createPool(), addLiquidity() a removeLiquidity(). Testy kontrolujú hlavný chod funkcie (teda beh funkcie pri ukončení bez chýb) a edge case funkcií, ktoré nastávajú pri nedodržaní podmienok vo vnútri require() klauzúl.

File	% Stmt	% Branch	% Funcs	% Lines	Uncovered Lines
<code>contracts\Lock.sol</code>	63.72	57.84	68.18	65.56	
<code>exchange.sol</code>	52.33	44.29	36.36	53.98	... 274,276,277
<code>token.sol</code>	100	84.62	100	100	
<code>contracts\interfaces\IERC20.sol</code>	100	100	100	100	
All files	63.72	57.84	68.18	65.56	

Výsledky coverage testov generovaných pomocou frameworku solidity coverage.

Spustenie testov:

Testy sú uložené vo vnútri priečinku test. Testy sú uložené vo vnútri súborov Token.js a Exchange.js. Pri tvorbe testov sme využili solidity coverage. Na spustenie všetkých testov treba v terminály vŕať dovnútra priečinku test a zavolať príkaz „npx hardhat coverage“.

Záver:

Na zadanie sme si vyskúšali programovať v jazyku solidity a vytvoriť a deploynúť si vlastné smart kontrakty v kontexte simulovaného prostredia blockchainu. Ďalej sme pochopili komunikácií medzi javascriptom a blockchainom. Vďaka tomu sme získali hlbší prehľad fungovaniu webových rozhraní pre komunikáciu s blockchainom. Zadanie sme si veľmi užili, ľubilo sa nám pracovať v dvojici.