Criando sua própria Criptomoeda

Sumário

[A moeda 2](#_Toc487534706)

[O código 2](#_Toc487534707)

[Token mínimo válido 4](#_Toc487534708)

[Entendendo o código 5](#_Toc487534709)

[Como fazer o Deploy? 9](#_Toc487534710)

[Melhorando seu Token 13](#_Toc487534711)

[Administrador 13](#_Toc487534712)

[Central 15](#_Toc487534713)

[Ativos 16](#_Toc487534714)

[Venda e Compra 16](#_Toc487534715)

[Autorização 18](#_Toc487534716)

[Proof Of Work 19](#_Toc487534717)

[Propriedades de Blocos e Transações 19](#_Toc487534718)

[Melhorando a moeda 22](#_Toc487534719)

[Código completo 22](#_Toc487534720)

[Deploying 26](#_Toc487534721)

[Usando sua moeda 29](#_Toc487534722)

[E agora? 30](#_Toc487534723)

# A moeda

Vamos criar um token digital. Tokens no ecossistema Ethereum podem representar qualquer bem fungível negociável: moedas, pontos de fidelidade, certificados de ouro, IOUs, itens de jogos, etc. Como todos os tokens implementam algumas características básicas de uma maneira padrão, isso também significa que seu token será instantaneamente compatível com a carteira Ethereum e qualquer outro cliente ou contrato que use os mesmos padrões.

## O código

Copie e cole o Código abaixo:

**pragma solidity ^0.4.8;**

**contract tokenRecipient { function receiveApproval(address \_from, uint256 \_value, address \_token, bytes \_extraData); }**

**contract MyToken {**

**/\* Public variables of the token \*/**

**string public standard = 'Token 0.1';**

**string public name;**

**string public symbol;**

**uint8 public decimals;**

**uint256 public totalSupply;**

**/\* This creates an array with all balances \*/**

**mapping (address => uint256) public balanceOf;**

**mapping (address => mapping (address => uint256)) public allowance;**

**/\* This generates a public event on the blockchain that will notify clients \*/**

**event Transfer(address indexed from, address indexed to, uint256 value);**

**/\* This notifies clients about the amount burnt \*/**

**event Burn(address indexed from, uint256 value);**

**/\* Initializes contract with initial supply tokens to the creator of the contract \*/**

**function MyToken(**

**uint256 initialSupply,**

**string tokenName,**

**uint8 decimalUnits,**

**string tokenSymbol**

**) {**

**balanceOf[msg.sender] = initialSupply; // Give the creator all initial tokens**

**totalSupply = initialSupply; // Update total supply**

**name = tokenName; // Set the name for display purposes**

**symbol = tokenSymbol; // Set the symbol for display purposes**

**decimals = decimalUnits; // Amount of decimals for display purposes**

**}**

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**if (\_to == 0x0) throw; // Prevent transfer to 0x0 address. Use burn() instead**

**if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

**balanceOf[msg.sender] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**Transfer(msg.sender, \_to, \_value); // Notify anyone listening that this transfer took place**

**}**

**/\* Allow another contract to spend some tokens in your behalf \*/**

**function approve(address \_spender, uint256 \_value)**

**returns (bool success) {**

**allowance[msg.sender][\_spender] = \_value;**

**return true;**

**}**

**/\* Approve and then communicate the approved contract in a single tx \*/**

**function approveAndCall(address \_spender, uint256 \_value, bytes \_extraData)**

**returns (bool success) {**

**tokenRecipient spender = tokenRecipient(\_spender);**

**if (approve(\_spender, \_value)) {**

**spender.receiveApproval(msg.sender, \_value, this, \_extraData);**

**return true;**

**}**

**}**

**/\* A contract attempts to get the coins \*/**

**function transferFrom(address \_from, address \_to, uint256 \_value) returns (bool success) {**

**if (\_to == 0x0) throw; // Prevent transfer to 0x0 address. Use burn() instead**

**if (balanceOf[\_from] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

**if (\_value > allowance[\_from][msg.sender]) throw; // Check allowance**

**balanceOf[\_from] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**allowance[\_from][msg.sender] -= \_value;**

**Transfer(\_from, \_to, \_value);**

**return true;**

**}**

**function burn(uint256 \_value) returns (bool success) {**

**if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough**

**balanceOf[msg.sender] -= \_value; // Subtract from the sender**

**totalSupply -= \_value; // Updates totalSupply**

**Burn(msg.sender, \_value);**

**return true;**

**}**

**function burnFrom(address \_from, uint256 \_value) returns (bool success) {**

**if (balanceOf[\_from] < \_value) throw; // Check if the sender has enough**

**if (\_value > allowance[\_from][msg.sender]) throw; // Check allowance**

**balanceOf[\_from] -= \_value; // Subtract from the sender**

**totalSupply -= \_value; // Updates totalSupply**

**Burn(\_from, \_value);**

**return true;**

**}**

**}**

## Token mínimo válido

O contrato token é bastante complexo. Mas, em essência, um token muito básico resume-se a isso:

**contract MyToken {**

**/\* This creates an array with all balances \*/**

**mapping (address => uint256) public balanceOf;**

**/\* Initializes contract with initial supply tokens to the creator of the contract \*/**

**function MyToken(**

**uint256 initialSupply**

**) {**

**balanceOf[msg.sender] = initialSupply; // Give the creator all initial tokens**

**}**

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

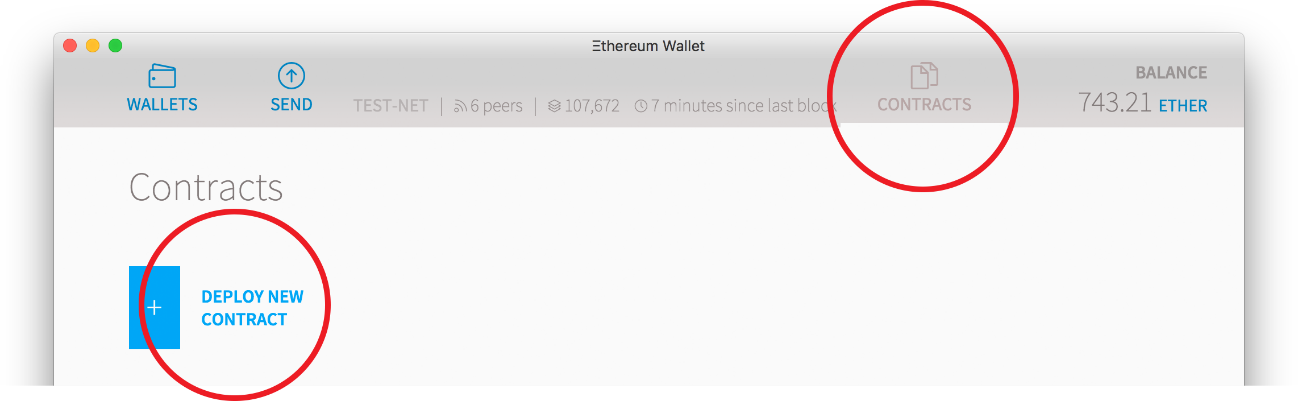
**balanceOf[msg.sender] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**}**

**}**

## Entendendo o código



Então, vamos começar com o básico. Abra o aplicativo **Wallet**, vá para a guia *Contracts* e, em seguida, *Deploy New Contract*. No campo de texto código de *Solidity Contract Source code*, digite o código abaixo:

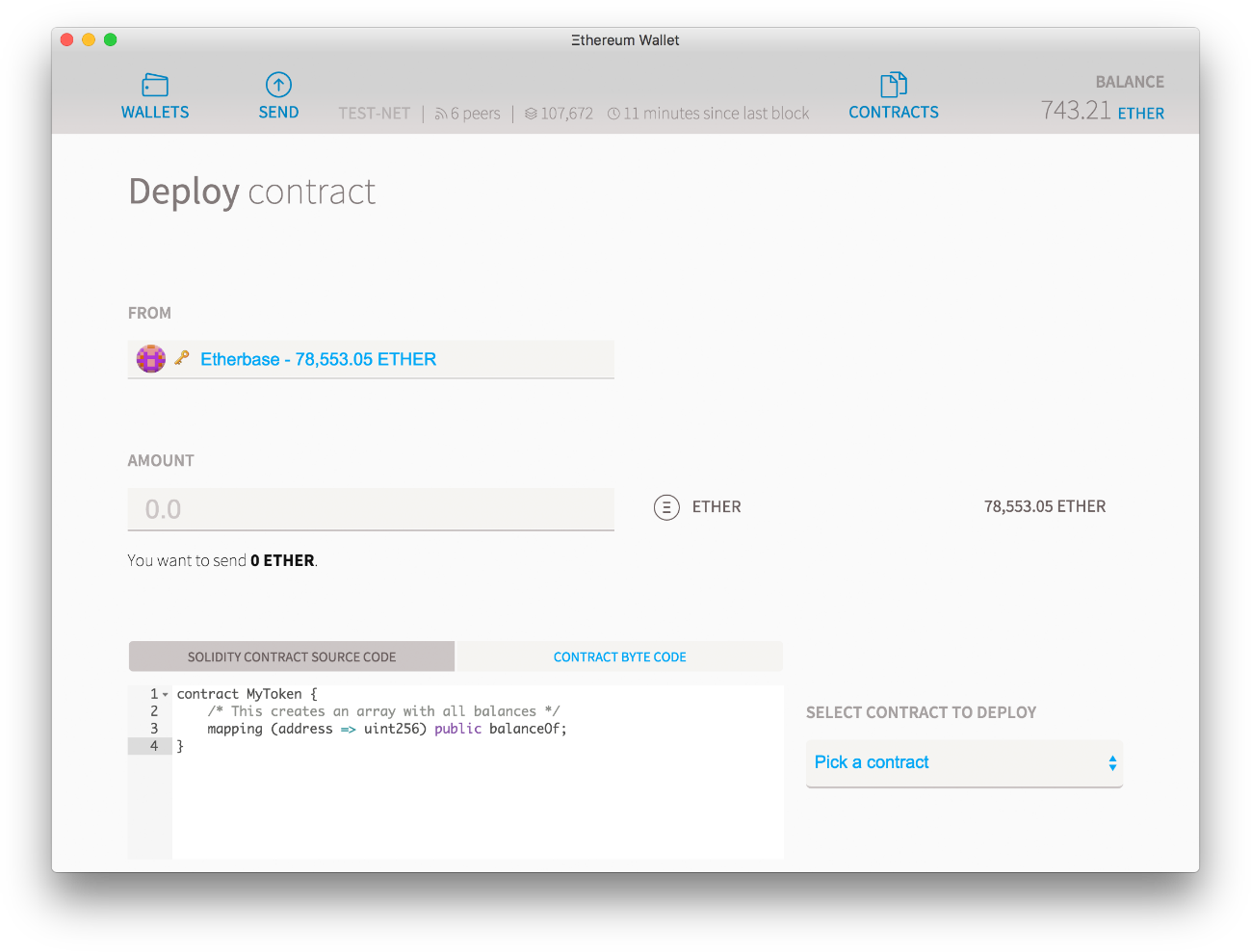
**contract MyToken {**

**/\* This creates an array with all balances \*/**

**mapping (address => uint256) public balanceOf;**

**}**

Um mapeamento significa uma matriz associativa, onde você associa endereços com saldos. Os endereços estão no formato Ethereum hexadecimal básico, enquanto os saldos são inteiros, variando de 0 a 115 quattuorvigintillion. Se você não sabe o quanto é um quattuorvigintillion, são muitos vigintillions mais do que tudo o que você está planejando usar seus tokens. A chave *pública* (public) significa que esta variável será acessível por qualquer pessoa na Blockchain, o que significa que todos os saldos são públicos (como eles precisam ser, para que os clientes possam exibi-los).



Se você publicou seu contrato imediatamente, funcionaria, mas não seria muito útil: seria um contrato que poderia consultar o equilíbrio de sua moeda para qualquer endereço, mas como você nunca criou uma única moeda, cada uma delas iria Retornar 0. Então, vamos criar alguns tokens na inicialização. Adicione este código abaixo.

**function MyToken() {**

**balanceOf[msg.sender] = 21000000;**

**}**

Observe que a *função MyToken* tem o mesmo nome que o contrato *MyToken*. Isso é muito importante e se você renomear um, você deve renomear o outro também: esta é uma função de inicialização especial que é executada apenas uma vez e uma vez somente quando o contrato é carregado pela primeira vez na rede. Esta função irá definir o saldo do *msg.sender*, o usuário que implantou o contrato, com um saldo de 21 milhões.

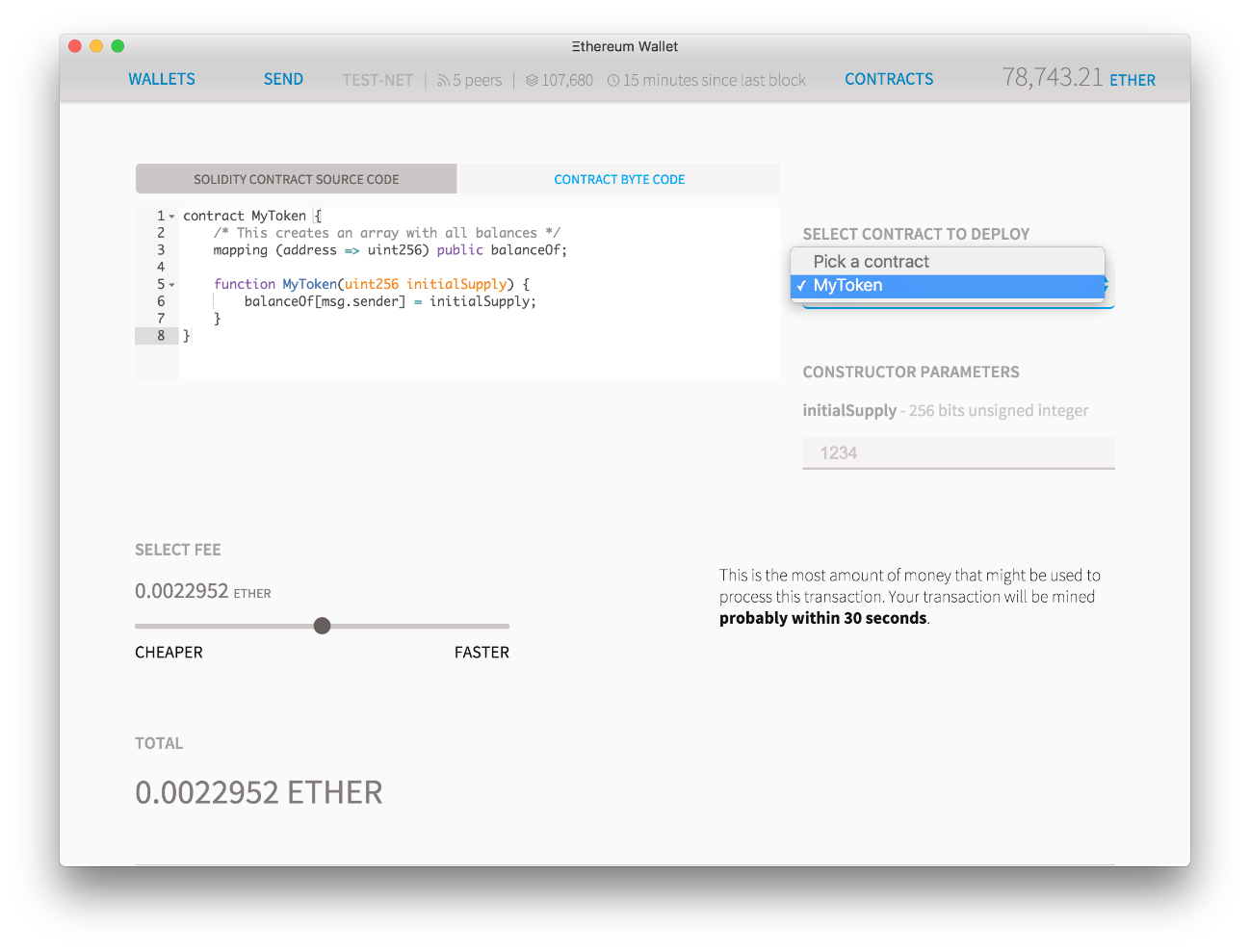
A escolha de 21 milhões foi aleatória, e você pode mudá-lo para qualquer coisa que desejar no código, mas há uma maneira melhor: em vez disso, forneça-o como um parâmetro para a função, assim:

**function MyToken(uint256 initialSupply) {**

**balanceOf[msg.sender] = initialSupply;**

**}**

Dê uma olhada na coluna direita, além do contrato e você verá um drop down, escrito, *escolha um contrato*. Selecione o contrato "MyToken" e você verá que agora ele mostra uma seção chamada *Parâmetros do Construtor (Constructor parameters)*. Estes são parâmetros variáveis para o seu token, para que você possa reutilizar o mesmo código e apenas alterar essas variáveis no futuro.



Agora você tem um contrato funcional que criou saldos de tokens, mas como não há nenhuma função para movê-lo, tudo o que faz é permanecer na mesma conta. Então, vamos implementá-lo agora. Escreva o seguinte código:

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**/\* Add and subtract new balances \*/**

**balanceOf[msg.sender] -= \_value;**

**balanceOf[\_to] += \_value;**

**}**

Esta é uma função muito direta: possui um destinatário e um valor como o parâmetro e sempre que alguém o chama, vai subtrair o valor do saldo e o adicionará ao saldo. De imediato, há um problema óbvio: o que acontece se a pessoa quiser enviar mais do que possui? Uma vez que não queremos lidar com dívidas neste contrato específico, simplesmente vamos fazer uma verificação rápida e se o remetente não tiver fundos suficientes, a execução do contrato simplesmente irá parar.

Para parar a execução de um contrato, você pode usar **return** ou **throw**. O primeiro custará menos gas, mas pode ser mais trabalhoso, pois as alterações que você fez no contrato até agora serão mantidas. Por outro lado, 'throw' cancelará toda a execução do contrato, reverterá quaisquer alterações que a transação possa ter feito e o remetente perderá todo o Ether que enviou para o gas. Mas, como o Wallet pode detectar que um contrato vai jogar, ele sempre mostra um alerta, impedindo assim que qualquer ether seja gasto.

**function transfer(address \_to, uint256 \_value) {**

**/\* Check if sender has balance and for overflows \*/**

**if (balanceOf[msg.sender] < \_value || balanceOf[\_to] + \_value < balanceOf[\_to])**

**throw;**

**/\* Add and subtract new balances \*/**

**balanceOf[msg.sender] -= \_value;**

**balanceOf[\_to] += \_value;**

**}**

Agora, tudo o que falta é termos algumas informações básicas sobre o contrato. No futuro próximo, isso pode ser tratado por um registro de token, mas agora vamos adicioná-los diretamente ao contrato:

**string public name;**

**string public symbol;**

**uint8 public decimals;**

E agora, atualizamos a **função do construtor** para permitir que todas essas variáveis sejam configuradas no início:

**/\* Initializes contract with initial supply tokens to the creator of the contract \*/**

**function MyToken(uint256 initialSupply, string tokenName, uint8 decimalUnits, string tokenSymbol) {**

**balanceOf[msg.sender] = initialSupply; // Give the creator all initial tokens**

**name = tokenName; // Set the name for display purposes**

**symbol = tokenSymbol; // Set the symbol for display purposes**

**decimals = decimalUnits; // Amount of decimals for display purposes**

**}**

Finalmente, agora precisamos de algo chamado **Events**. Estas são funções especiais e vazias que você chama para ajudar clientes como Ethereum Wallet a acompanhar as atividades que acontecem no contrato. Os eventos devem começar com uma letra maiúscula. Adicione esta linha no início do contrato para declarar o evento:

**event Transfer(address indexed from, address indexed to, uint256 value);**

E então você só precisa adicionar essas duas linhas dentro da função "**Transfer**":

**/\* Notify anyone listening that this transfer took place \*/**

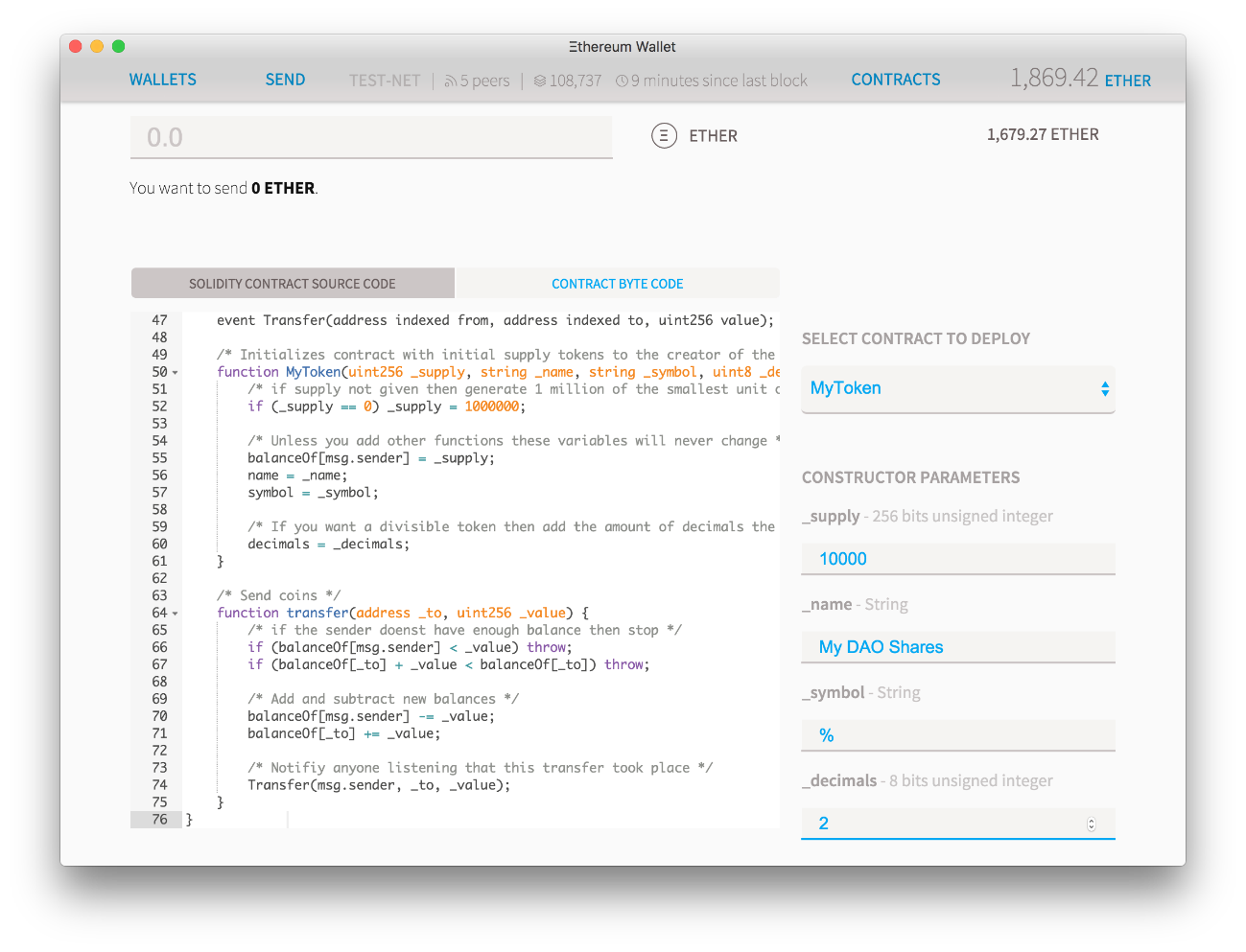
**Transfer(msg.sender, \_to, \_value);**

E agora o seu token está pronto!

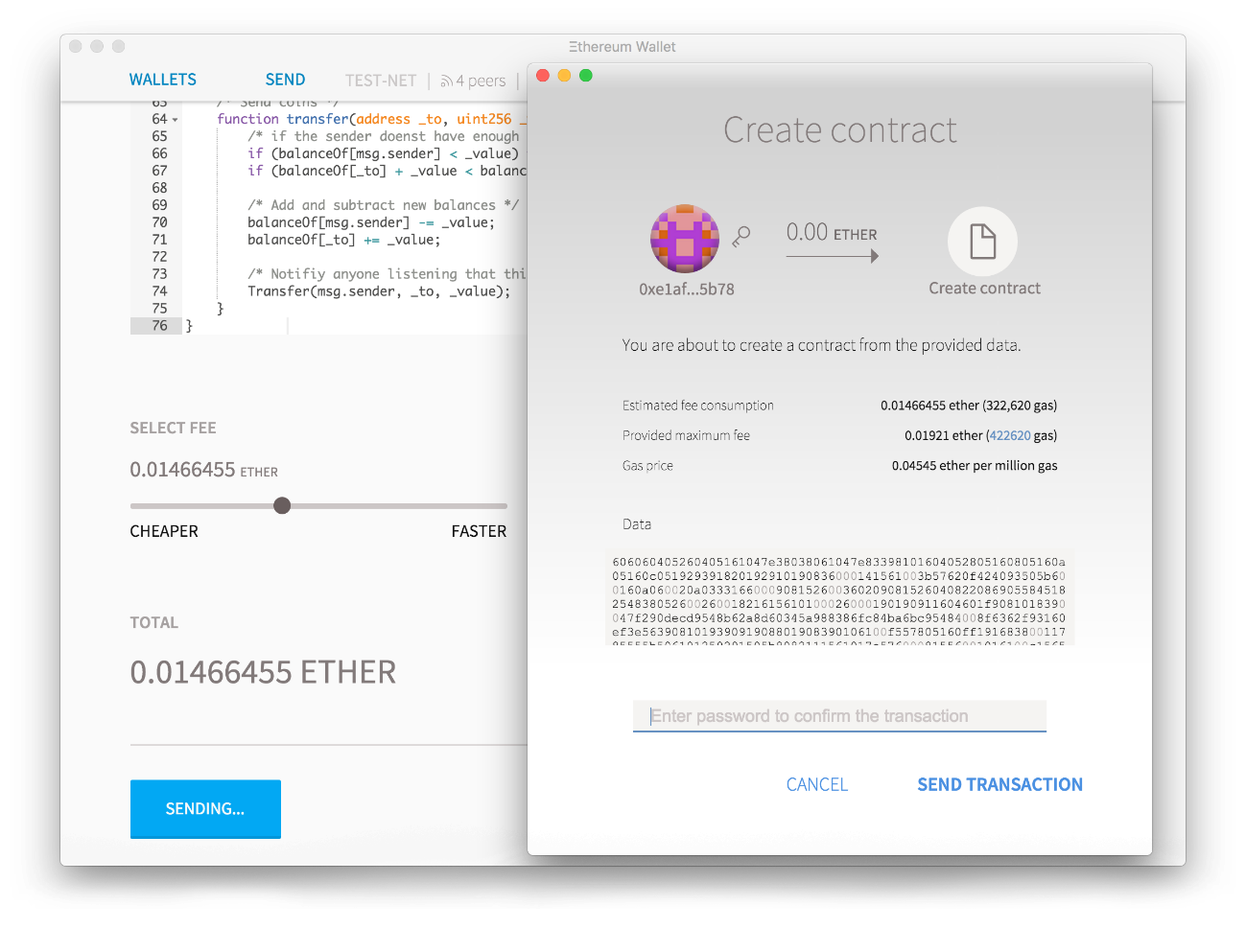
## Como fazer o Deploy?

Abra a Ethereum Wallet, vá para a aba de contratos e clique em "deploy new contract".

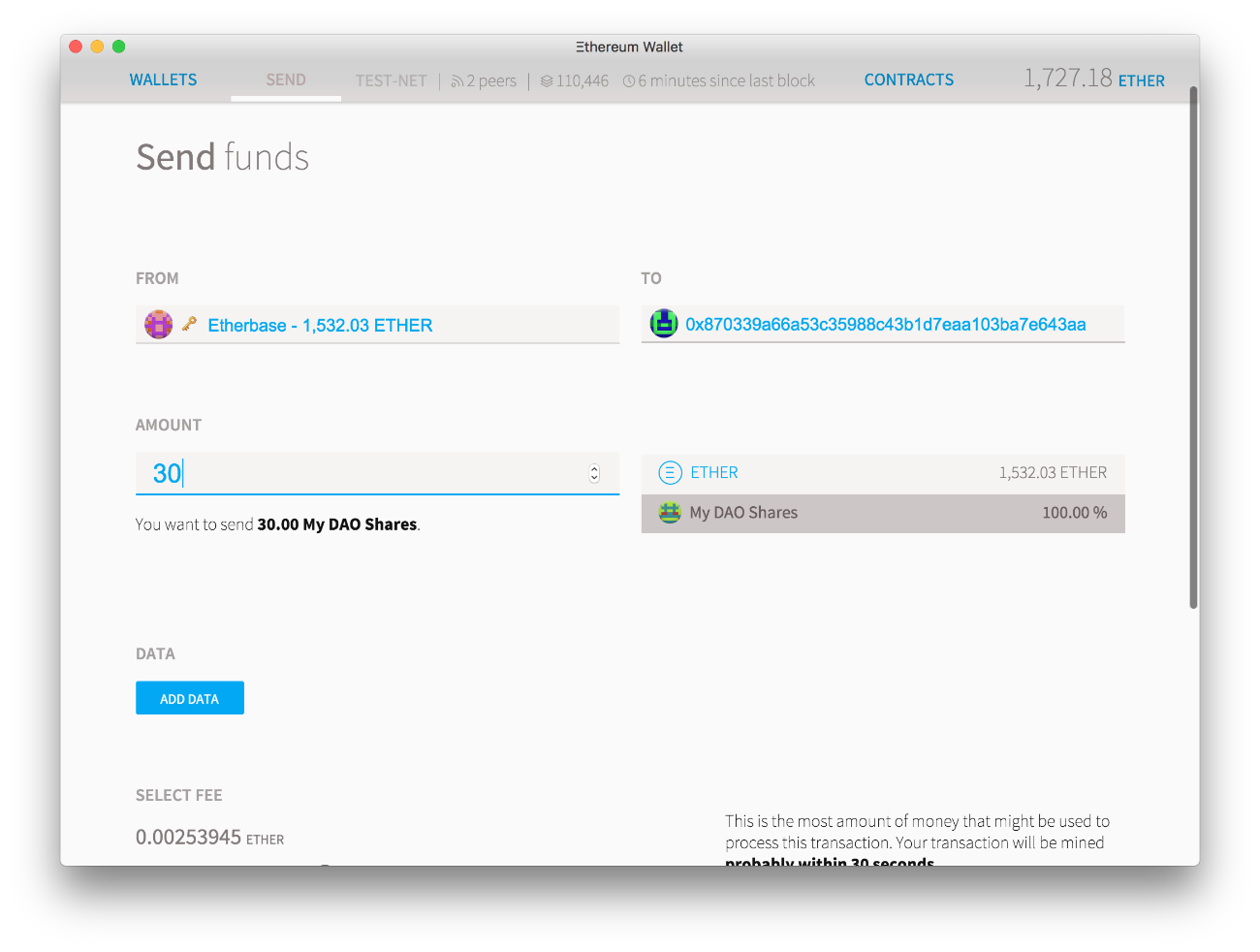
Agora, obtenha a fonte de token de cima e cole-a no "Solidity Source field". Se o código compilar sem qualquer erro, você deve ver um menu "pick a contract" no lado direito. Selecione o contrato "MyToken". Na coluna da direita você verá todos os parâmetros que você precisa para personalizar seu próprio token. Você pode ajustá-los como quiser, mas, para o propósito deste tutorial, recomendamos que você escolha esses parâmetros: 10 000 como o fornecimento, qualquer nome desejado, "%" para um símbolo e 2 casas decimais. Seu aplicativo deve estar assim:



Vá até o final da página e você verá uma estimativa do custo de cálculo desse contrato e você pode selecionar uma taxa sobre quanto é que você está disposto a pagar por isso. Qualquer excesso de ether que você não gastará será retornado para você, para que você possa deixar as configurações padrão, se desejar. Pressione "deploy", digite a senha da sua conta e espere alguns segundos para que sua transação seja realizada.

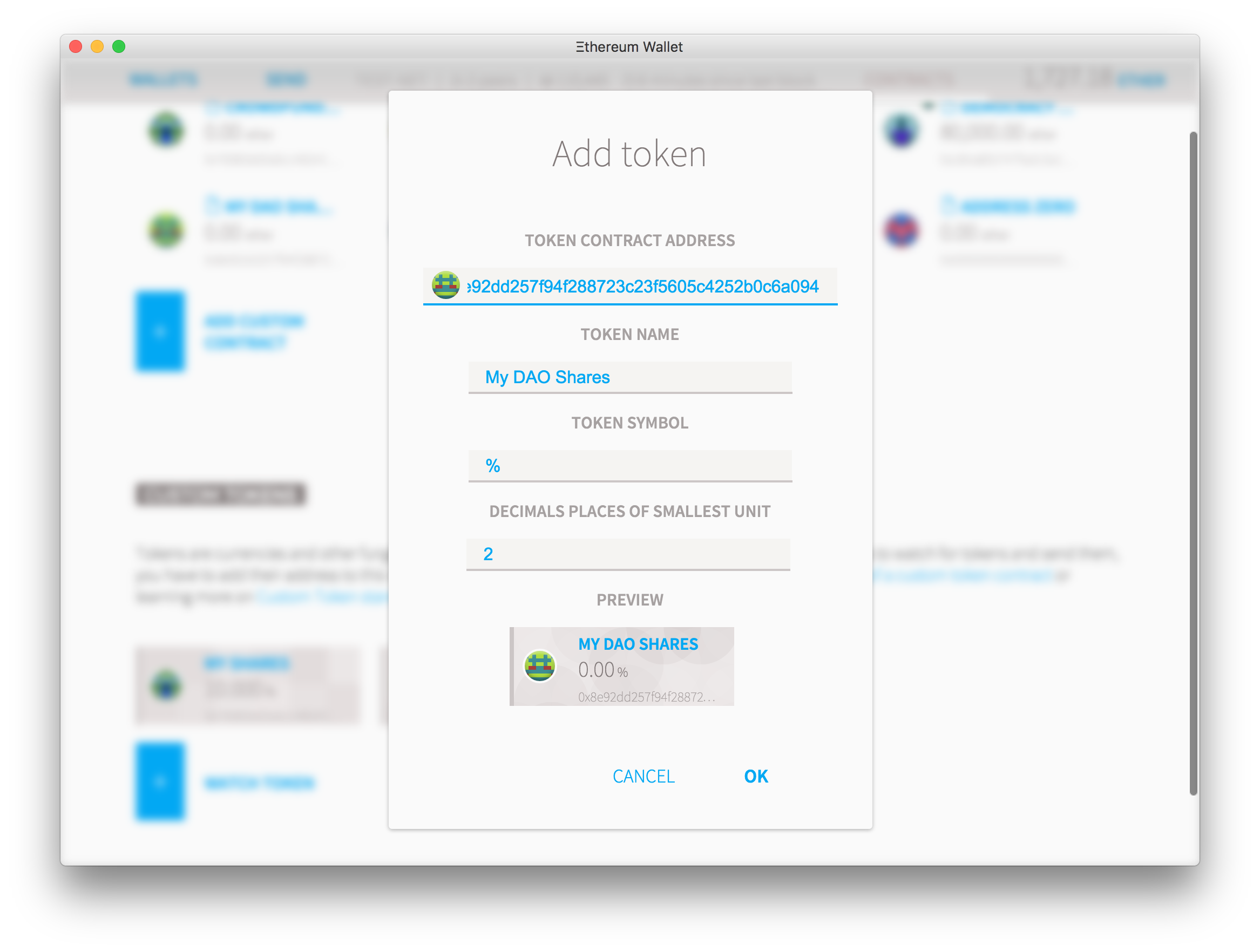


Você será redirecionado para a página inicial onde você pode ver sua transação aguardando confirmações. Clique na conta chamada "Etherbase" (sua conta principal) e, depois de mais de um minuto, você deve ver que sua conta mostrará que você possui 100% das ações que você acabou de criar. Para enviar alguns para alguns amigos: selecione "send" e escolha a moeda que deseja enviar (ether ou o seu próprio token), cole o endereço do seu amigo no campo "to" e pressione "send".



Se você o enviar para um amigo, eles ainda não verão nada na carteira. Isso ocorre porque a carteira não possui tokens que conhece e você deve adicioná-los manualmente. Agora vá para a aba "Contracts" e você deve ver um link para seu contrato recém-criado. Clique nele para ir à sua página. Uma vez que esta é uma página de contrato muito simples, não há muito a fazer aqui, basta clicar em "copy address" e colar o endereço do contrato em um editor de texto, você precisará dele em breve.

Para adicionar um token, vá para a página de contratos e, em seguida, clique em "Watch Token" (Assinar Token). Aparecerá um pop-up e você só precisa colar o endereço do contrato. O nome do token, o símbolo e o número decimal devem ser preenchidos automaticamente, mas você não pode colocar o que quiser (isso afetará somente a forma como ele é exibido em sua carteira). Depois de fazer isso, você mostrará automaticamente qualquer saldo que você tenha desse token e você poderá enviá-lo para qualquer outra pessoa.



Agora você tem seu próprio token! Os tokens por si só podem ser úteis como troca de valor nas comunidades locais, formas de acompanhar as horas trabalhadas ou outros programas de fidelidade, etc.

# Melhorando seu Token

Você pode implantar todo o seu token de criptografia sem tocar uma linha de código, mas a magia real acontece quando você começa a personalizá-la. As seções a seguir serão sugestões sobre as funções que você pode adicionar ao seu token para que ele atenda melhor suas necessidades.

## Administrador

Todos os dapps são totalmente descentralizados por padrão, mas isso não significa que eles não podem ter algum tipo de administrador central, se você quiser. Talvez você queira a capacidade de emitir mais moedas ou talvez você queira banir algumas pessoas. Você pode adicionar qualquer desses recursos, mas só você pode adicioná-los, então todos os titulares de token sempre saberão exatamente as regras do jogo antes de decidirem possuir um.

Para que isso aconteça, você precisa de um controlador central desta moeda. Esta poderia ser uma conta simples, mas também poderia ser um contrato e, portanto, a decisão de criar mais tokens dependerá do contrato: se é uma organização democrática que pode votar, ou talvez seja apenas uma maneira de limitar a autonomia do proprietário token.

Para fazer isso, aprenderemos uma propriedade muito útil de contratos: herança (inheritance). A herança permite que um contrato adquira propriedades de um contrato-mãe, sem ter que os redefinir. Isso torna o código mais limpo e mais fácil de reutilizar. Adicione este código à primeira linha do seu código, **antes do contrato MyToken {**.

**contract owned {**

**address public owner;**

**function owned() {**

**owner = msg.sender;**

**}**

**modifier onlyOwner {**

**if (msg.sender != owner) throw;**

**\_;**

**}**

**function transferOwnership(address newOwner) onlyOwner {**

**owner = newOwner;**

**}**

**}**

Isso cria um contrato muito básico que não faz nada além de definir algumas funções genéricas sobre um contrato que pode ser "de propriedade". Agora, o próximo passo é apenas adicionar o texto é de propriedade do seu contrato:

**contract MyToken is owned {**

**/\* the rest of the contract as usual \*/**

Isso significa que todas as funções dentro do **MyToken** agora podem acessar o proprietário da variável *owner* e o modificador apenas o **Proprietário**. O contrato também possui uma função para transferir a propriedade. Como pode ser interessante configurar o proprietário do contrato na inicialização, você também pode adicionar isso à função de construtor:

**function MyToken(**

**uint256 initialSupply,**

**string tokenName,**

**uint8 decimalUnits,**

**string tokenSymbol,**

**address centralMinter**

**) {**

**if(centralMinter != 0 ) owner = centralMinter;**

## Central

Suponha que você deseja que a quantidade de moedas em circulação mude. Este é o caso quando seus tokens realmente representam um recurso off blockchain (como certificados de ouro ou moedas do governo) e você quer que o inventário reflita o real. Isso também pode ser o caso quando os detentores de moeda esperam algum controle do preço do token e querem emitir ou remover tokens da circulação.

Primeiro, precisamos adicionar uma variável para armazenar o **totalSupply** e atribuí-lo em nossa função de construtor.

**contract MyToken {**

**uint256 public totalSupply;**

**function MyToken(...) {**

**totalSupply = initialSupply;**

**...**

**}**

**...**

**}**

Agora vamos adicionar uma nova função, finalmente, que permitirá ao proprietário criar novos tokens:

**function mintToken(address target, uint256 mintedAmount) onlyOwner {**

**balanceOf[target] += mintedAmount;**

**totalSupply += mintedAmount;**

**Transfer(0, owner, mintedAmount);**

**Transfer(owner, target, mintedAmount);**

**}**

Observe o modificador **apenas o Proprietário** (**onlyOwner)** no final do nome da função. Isso significa que esta função será reescrita na compilação para herdar o código do modificador **apenas o Proprietário** que tínhamos definido anteriormente. O código desta função será inserido onde há um sublinhado na função modificadora, o que significa que esta função particular só pode ser chamada pela conta que é definida como o proprietário. Basta adicionar isso a um contrato com um modificador do proprietário e você poderá criar mais moedas.

## Ativos

Dependendo do seu caso de uso, você poderá ter alguns obstáculos regulatórios sobre quem pode e não pode usar seus tokens. Para que isso aconteça, você pode adicionar um parâmetro que permita ao proprietário do contrato congelar ou descongelar ativos.

Adicione esta variável e funcione em qualquer lugar dentro do contrato. Você pode colocá-los em qualquer lugar, mas para uma boa prática, recomendamos que você coloque os mapeamentos com os outros mapeamentos e eventos com os outros eventos.

**mapping (address => bool) public frozenAccount;**

**event FrozenFunds(address target, bool frozen);**

**function freezeAccount(address target, bool freeze) onlyOwner {**

**frozenAccount[target] = freeze;**

**FrozenFunds(target, freeze);**

**}**

Com este código, todas as contas são descongeladas por padrão, mas o proprietário pode definir qualquer uma delas em um estado de congelamento, ligando a **conta Freeze**. Infelizmente, o congelamento não tem efeito prático, porque não adicionamos nada à função de transferência. Estamos mudando isso agora:

**function transfer(address \_to, uint256 \_value) {**

**if (frozenAccount[msg.sender]) throw;**

Agora, qualquer conta que esteja congelada ainda terá seus fundos intactos, mas não poderá movê-los. Todas as contas são descongeladas por padrão até congelá-las, mas você pode facilmente reverter esse comportamento para uma whitelist onde você precisa aprovar manualmente todas as contas. Basta renomear **frozenAccount** para **approvedAccount**

**if (!approvedAccount[msg.sender]) throw;**

## Venda e Compra

Até agora, você confiou em utilidade e confiança para avaliar o seu token. Mas se você quiser, você pode fazer com que o valor do token seja apoiado pelo ether (ou outros tokens), criando um fundo que vende e compra automaticamente ao valor de mercado.

Primeiro, vamos definir o preço para comprar e vender:

**uint256 public sellPrice;**

**uint256 public buyPrice;**

**function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner {**

**sellPrice = newSellPrice;**

**buyPrice = newBuyPrice;**

**}**

Isso é aceitável por um preço que não muda com muita frequência, pois cada nova mudança de preço exigirá que você execute uma transação e gaste um pouco de ether. Se você quiser ter um preço flutuante constante, recomendamos investigar os feeds de dados padrão <https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs#data-feeds>

O próximo passo é fazer as funções de compra e venda:

**function buy() payable returns (uint amount){**

**amount = msg.value / buyPrice; // calculates the amount**

**if (balanceOf[this] < amount) throw; // checks if it has enough to sell**

**balanceOf[msg.sender] += amount; // adds the amount to buyer's balance**

**balanceOf[this] -= amount; // subtracts amount from seller's balance**

**Transfer(this, msg.sender, amount); // execute an event reflecting the change**

**return amount; // ends function and returns**

**}**

**function sell(uint amount) returns (uint revenue){**

**if (balanceOf[msg.sender] < amount ) throw; // checks if the sender has enough to sell**

**balanceOf[this] += amount; // adds the amount to owner's balance**

**balanceOf[msg.sender] -= amount; // subtracts the amount from seller's balance**

**revenue = amount \* sellPrice;**

**if (!msg.sender.send(revenue)) { // sends ether to the seller: it's important**

**throw; // to do this last to prevent recursion attacks**

**} else {**

**Transfer(msg.sender, this, amount); // executes an event reflecting on the change**

**return revenue; // ends function and returns**

**}**

**}**

Observe que isso não criará novos tokens, mas alterará o saldo que o contrato possui. O contrato pode armazenar seus próprios tokens e ether o proprietário do contrato, enquanto ele pode definir preços ou, em alguns casos, criar novos tokens (se aplicável), não pode tocar os tokens ou o ether do banco. A única maneira como esse contrato pode mover fundos é vendê-los e comprá-los.

Nota: Comprar e vender "**preços**" não são estabelecidos em ether, mas em wei a moeda mínima do sistema (equivalente ao centavo no Euro e Dólar, ou o Satoshi em Bitcoin). Um éter é 1000000000000000000 wei. Então, ao ajustar os preços para o seu token em éter, adicione 18 zeros no final.

Ao criar o contrato, envie bastante moedas para que ele possa comprar todos os tokens no mercado, caso contrário seu contrato será insolvente e seus usuários não poderão vender seus tokens.

Os exemplos anteriores, é claro, descrevem um contrato com um único comprador e vendedor central, um contrato muito mais interessante permitiria um mercado onde qualquer um pode oferecer preços diferentes, ou talvez ele carregue os preços diretamente de uma fonte externa.

## Autorização

Toda vez que você faz uma transação no Ethereum, você precisa pagar uma taxa para o minerador do bloco que calculará o resultado do seu contrato inteligente. Embora isso possa mudar no futuro\*, pelo momento as taxas só podem ser pagas em ether e, portanto, todos os usuários de seus tokens precisam disso. Tokens em contas com um saldo menor do que as taxas estão presas até que o proprietário possa pagar a taxa necessária. Mas em alguns casos, você pode não querer que seus usuários pensem sobre ethereum, blockchain ou como obter ether, então uma possível abordagem teria sua moeda automaticamente recarregar o balanço do usuário assim que detecte que o saldo é perigosamente baixo.

\* <https://github.com/ethereum/EIPs/issues/28>

Para fazer isso, primeiro você precisa criar uma variável que contenha o valor limite e uma função para alterá-lo. Se você não conhece nenhum valor, ajuste-o para **5 finney (0,005 ether).**

**uint minBalanceForAccounts;**

**function setMinBalance(uint minimumBalanceInFinney) onlyOwner {**

**minBalanceForAccounts = minimumBalanceInFinney \* 1 finney;**

**}**

Em seguida, adicione esta linha à função de transferência para que o remetente seja reembolsado:

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**...**

**if(msg.sender.balance<minBalanceForAccounts)**

**sell((minBalanceForAccounts-msg.sender.balance)/sellPrice);**

**}**

Você também pode alterá-lo de modo que a taxa seja paga antecipadamente para o destinatário pelo remetente:

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**...**

**if(\_to.balance<minBalanceForAccounts)**

**\_to.send(sell((minBalanceForAccounts-\_to.balance)/sellPrice));**

**}**

Isso garantirá que nenhuma conta que receba o token tenha menos que o ether necessário para pagar as taxas.

## Proof Of Work

Existem algumas maneiras de amarrar seu suprimento de moeda com uma fórmula matemática. Uma das formas mais simples seria tornar-se uma "mineração fundida" (merged mining) com ether, o que significa que qualquer pessoa que encontre um bloqueio no Ethereum também receberia uma recompensa da sua moeda, dado que alguém chama a função de recompensa nesse bloco. Você pode fazê-lo usando a base de palavras-chave especial que se refere ao mineiro que encontra o bloco.

### Propriedades de Blocos e Transações

block.blockhash(uint blockNumber) returns (bytes32): hash of the given block - only works for 256 most recent blocks excluding current

block.coinbase (address): current block miner’s address

block.difficulty (uint): current block difficulty

block.gaslimit (uint): current block gaslimit

block.number (uint): current block number

block.timestamp (uint): current block timestamp as seconds since unix epoch

msg.data (bytes): complete calldata

msg.gas (uint): remaining gas

msg.sender (address): sender of the message (current call)

msg.sig (bytes4): first four bytes of the calldata (i.e. function identifier)

msg.value (uint): number of wei sent with the message

now (uint): current block timestamp (alias for block.timestamp)

tx.gasprice (uint): gas price of the transaction

tx.origin (address): sender of the transaction (full call chain)

**function giveBlockReward() {**

**balanceOf[block.coinbase] += 1;**

**}**

Também é possível adicionar uma fórmula matemática, para que qualquer pessoa que possa fazer a mineração pode ganhar uma recompensa. Neste próximo exemplo, você deve calcular a raiz cúbica do desafio atual e um direito e definir o próximo desafio:

**uint currentChallenge = 1; // Can you figure out the cubic root of this number?**

**function rewardMathGeniuses(uint answerToCurrentReward, uint nextChallenge) {**

**if (answerToCurrentReward\*\*3 != currentChallenge) throw; // If answer is wrong do not continue**

**balanceOf[msg.sender] += 1; // Reward the player**

**currentChallenge = nextChallenge; // Set the next challenge**

**}**

Claro que ao calcular raízes cúbicas pode ser difícil para alguém fazer de cabeça, eles são muito fáceis com uma calculadora, então esse jogo pode ser facilmente quebrado por um computador. Também desde que o último vencedor pode escolher o próximo desafio, eles podem escolher algo que eles conhecem e, portanto, não seria um jogo muito justo para outros jogadores. Existem tarefas que são fáceis para humanos, mas difíceis para computadores, mas geralmente são muito difíceis de codificar em scripts simples como esses. Em vez disso, um sistema mais justo deve ser um que seja muito difícil para um computador, mas fácil para um outro computador verificar. Um excelente candidato seria criar um desafio de hash onde o desafiante tem que gerar hashes de vários números até encontrar um que seja menor do que uma determinada dificuldade.

Esse processo foi proposto pela primeira vez por Adam Back em 1997 como Hashcash e depois foi implementado na Bitcoin por Satoshi Nakamoto como prova de trabalho em 2008. Ethereum foi lançado usando esse sistema para seu modelo de segurança, mas está planejando passar de uma segurança de Prova de Trabalho Modelo em uma prova mista de jogo e sistema de apostas chamado Casper.

<https://en.wikipedia.org/wiki/Hashcash>

<https://blog.ethereum.org/2015/12/28/understanding-serenity-part-2-casper/>

Mas se você gosta de Hashing como uma forma de emissão aleatória de moedas, você ainda pode criar sua própria moeda baseada em Ethereum que tenha uma prova de emissão de trabalho:

**bytes32 public currentChallenge; // The coin starts with a challenge**

**uint public timeOfLastProof; // Variable to keep track of when rewards were given**

**uint public difficulty = 10\*\*32; // Difficulty starts reasonably low**

**function proofOfWork(uint nonce){**

**bytes8 n = bytes8(sha3(nonce, currentChallenge)); // Generate a random hash based on input**

**if (n < bytes8(difficulty)) throw; // Check if it's under the difficulty**

**uint timeSinceLastProof = (now - timeOfLastProof); // Calculate time since last reward was given**

**if (timeSinceLastProof < 5 seconds) throw; // Rewards cannot be given too quickly**

**balanceOf[msg.sender] += timeSinceLastProof / 60 seconds; // The reward to the winner grows by the minute**

**difficulty = difficulty \* 10 minutes / timeSinceLastProof + 1; // Adjusts the difficulty**

**timeOfLastProof = now; // Reset the counter**

**currentChallenge = sha3(nonce, currentChallenge, block.blockhash(block.number-1)); // Save a hash that will be used as the next proof**

**}**

Altere também a **função Construtor** (aquele que tem o mesmo nome que o contrato, que é chamado no primeiro carregamento) para adicionar esta linha, então ajuste a dificuldade:

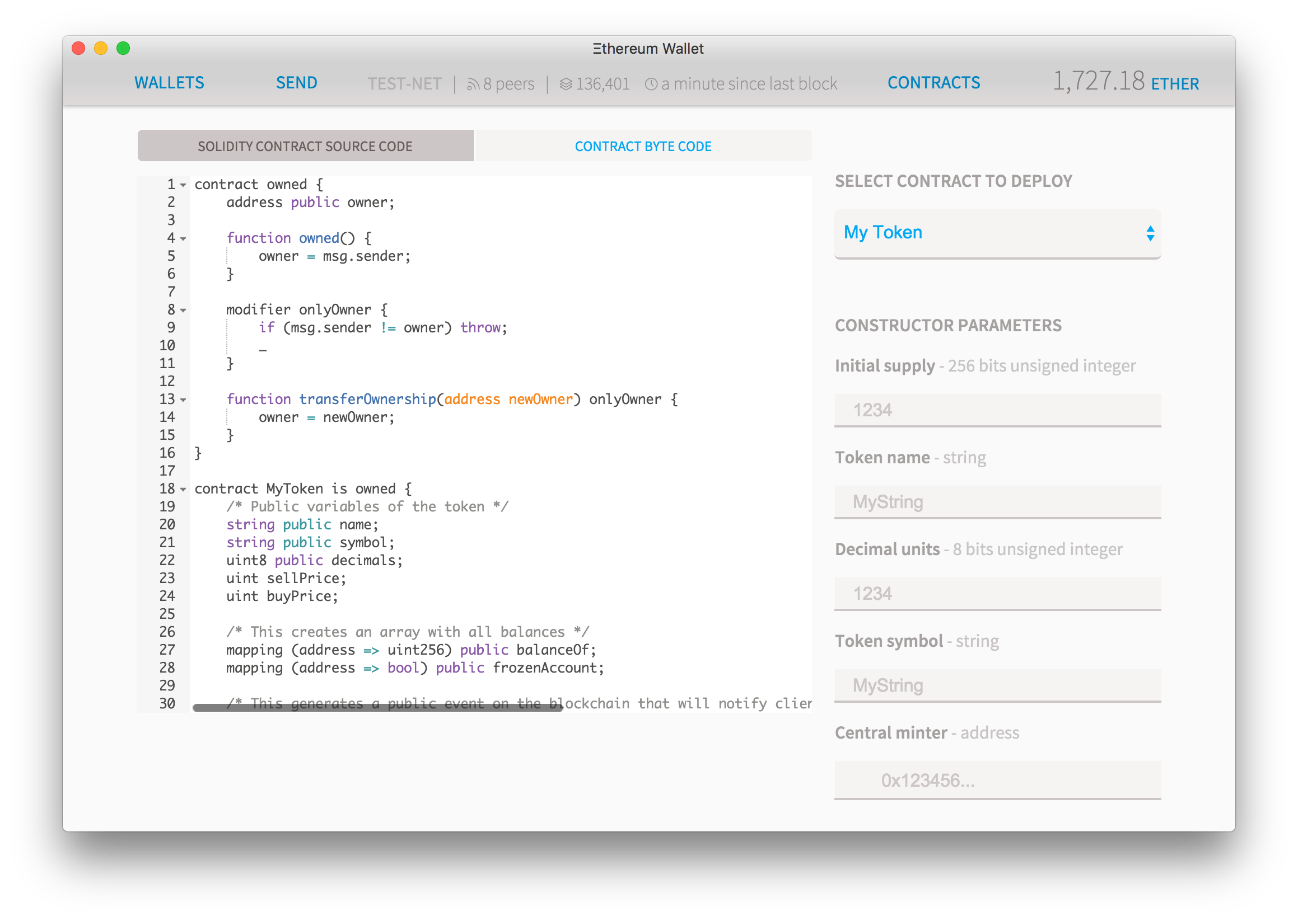
**timeOfLastProof = now;**

Uma vez que o contrato está online, selecione a função "Proof of Work", adicione o seu número favorito no campo **nonce** e tente executá-lo. Se a janela de confirmação retornar um aviso vermelho dizendo "Dados não podem ser executados" (Data can't be execute), volte e escolha outro número até encontrar um que permita que a transação avançar: esse processo é aleatório. Se você encontrar um, você receberá 1 token por cada minuto que tenha passado desde que a última recompensa foi dada, e então a dificuldade do desafio será ajustada para cima ou para baixo para atingir uma média de 10 minutos por recompensa.

Este processo de tentar encontrar o número que lhe dará uma recompensa é o que é chamado de **mineração**: se surgir dificuldade, pode ser muito difícil encontrar um número de sorte, mas sempre será fácil verificar que você encontrou um.

# Melhorando a moeda

Se você chegou até aqui o seu código deve estar igual a este:



## Código completo

**pragma solidity ^0.4.2;**

**contract owned {**

**address public owner;**

**function owned() {**

**owner = msg.sender;**

**}**

**modifier onlyOwner {**

**if (msg.sender != owner) throw;**

**\_;**

**}**

**function transferOwnership(address newOwner) onlyOwner {**

**owner = newOwner;**

**}**

**}**

**contract tokenRecipient { function receiveApproval(address \_from, uint256 \_value, address \_token, bytes \_extraData); }**

**contract token {**

**/\* Public variables of the token \*/**

**string public standard = 'Token 0.1';**

**string public name;**

**string public symbol;**

**uint8 public decimals;**

**uint256 public totalSupply;**

**/\* This creates an array with all balances \*/**

**mapping (address => uint256) public balanceOf;**

**mapping (address => mapping (address => uint256)) public allowance;**

**/\* This generates a public event on the blockchain that will notify clients \*/**

**event Transfer(address indexed from, address indexed to, uint256 value);**

**/\* Initializes contract with initial supply tokens to the creator of the contract \*/**

**function token(**

**uint256 initialSupply,**

**string tokenName,**

**uint8 decimalUnits,**

**string tokenSymbol**

**) {**

**balanceOf[msg.sender] = initialSupply; // Give the creator all initial tokens**

**totalSupply = initialSupply; // Update total supply**

**name = tokenName; // Set the name for display purposes**

**symbol = tokenSymbol; // Set the symbol for display purposes**

**decimals = decimalUnits; // Amount of decimals for display purposes**

**}**

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

**balanceOf[msg.sender] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**Transfer(msg.sender, \_to, \_value); // Notify anyone listening that this transfer took place**

**}**

**/\* Allow another contract to spend some tokens in your behalf \*/**

**function approve(address \_spender, uint256 \_value)**

**returns (bool success) {**

**allowance[msg.sender][\_spender] = \_value;**

**return true;**

**}**

**/\* Approve and then communicate the approved contract in a single tx \*/**

**function approveAndCall(address \_spender, uint256 \_value, bytes \_extraData)**

**returns (bool success) {**

**tokenRecipient spender = tokenRecipient(\_spender);**

**if (approve(\_spender, \_value)) {**

**spender.receiveApproval(msg.sender, \_value, this, \_extraData);**

**return true;**

**}**

**}**

**/\* A contract attempts to get the coins \*/**

**function transferFrom(address \_from, address \_to, uint256 \_value) returns (bool success) {**

**if (balanceOf[\_from] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

**if (\_value > allowance[\_from][msg.sender]) throw; // Check allowance**

**balanceOf[\_from] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**allowance[\_from][msg.sender] -= \_value;**

**Transfer(\_from, \_to, \_value);**

**return true;**

**}**

**/\* This unnamed function is called whenever someone tries to send ether to it \*/**

**function () {**

**throw; // Prevents accidental sending of ether**

**}**

**}**

**contract MyAdvancedToken is owned, token {**

**uint256 public sellPrice;**

**uint256 public buyPrice;**

**mapping (address => bool) public frozenAccount;**

**/\* This generates a public event on the blockchain that will notify clients \*/**

**event FrozenFunds(address target, bool frozen);**

**/\* Initializes contract with initial supply tokens to the creator of the contract \*/**

**function MyAdvancedToken(**

**uint256 initialSupply,**

**string tokenName,**

**uint8 decimalUnits,**

**string tokenSymbol**

**) token (initialSupply, tokenName, decimalUnits, tokenSymbol) {}**

**/\* Send coins \*/**

**function transfer(address \_to, uint256 \_value) {**

**if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

**if (frozenAccount[msg.sender]) throw; // Check if frozen**

**balanceOf[msg.sender] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**Transfer(msg.sender, \_to, \_value); // Notify anyone listening that this transfer took place**

**}**

**/\* A contract attempts to get the coins \*/**

**function transferFrom(address \_from, address \_to, uint256 \_value) returns (bool success) {**

**if (frozenAccount[\_from]) throw; // Check if frozen**

**if (balanceOf[\_from] < \_value) throw; // Check if the sender has enough**

**if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows**

**if (\_value > allowance[\_from][msg.sender]) throw; // Check allowance**

**balanceOf[\_from] -= \_value; // Subtract from the sender**

**balanceOf[\_to] += \_value; // Add the same to the recipient**

**allowance[\_from][msg.sender] -= \_value;**

**Transfer(\_from, \_to, \_value);**

**return true;**

**}**

**function mintToken(address target, uint256 mintedAmount) onlyOwner {**

**balanceOf[target] += mintedAmount;**

**totalSupply += mintedAmount;**

**Transfer(0, this, mintedAmount);**

**Transfer(this, target, mintedAmount);**

**}**

**function freezeAccount(address target, bool freeze) onlyOwner {**

**frozenAccount[target] = freeze;**

**FrozenFunds(target, freeze);**

**}**

**function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner {**

**sellPrice = newSellPrice;**

**buyPrice = newBuyPrice;**

**}**

**function buy() payable {**

**uint amount = msg.value / buyPrice; // calculates the amount**

**if (balanceOf[this] < amount) throw; // checks if it has enough to sell**

**balanceOf[msg.sender] += amount; // adds the amount to buyer's balance**

**balanceOf[this] -= amount; // subtracts amount from seller's balance**

**Transfer(this, msg.sender, amount); // execute an event reflecting the change**

**}**

**function sell(uint256 amount) {**

**if (balanceOf[msg.sender] < amount ) throw; // checks if the sender has enough to sell**

**balanceOf[this] += amount; // adds the amount to owner's balance**

**balanceOf[msg.sender] -= amount; // subtracts the amount from seller's balance**

**if (!msg.sender.send(amount \* sellPrice)) { // sends ether to the seller. It's important**

**throw; // to do this last to avoid recursion attacks**

**} else {**

**Transfer(msg.sender, this, amount); // executes an event reflecting on the change**

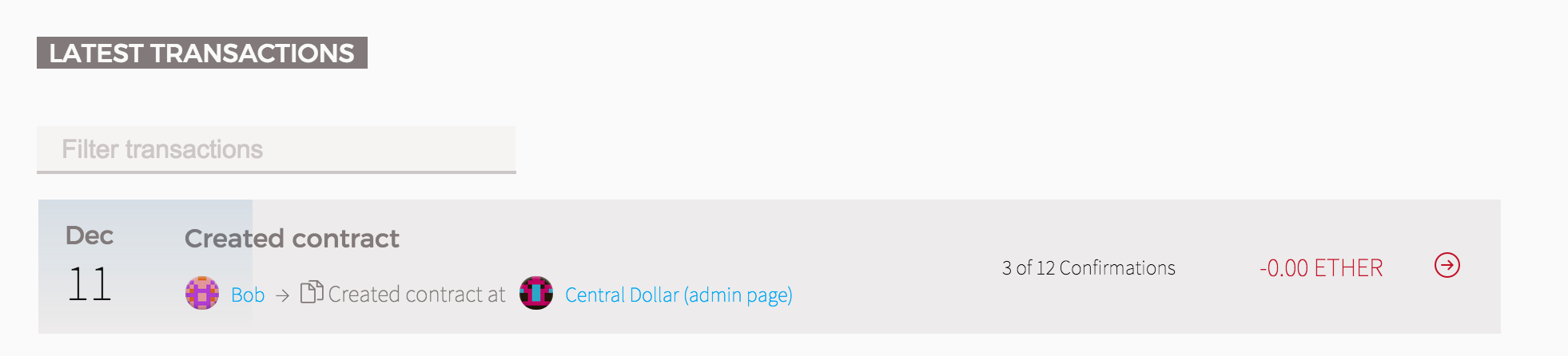
**}**

**}**

**}**

## Deploying

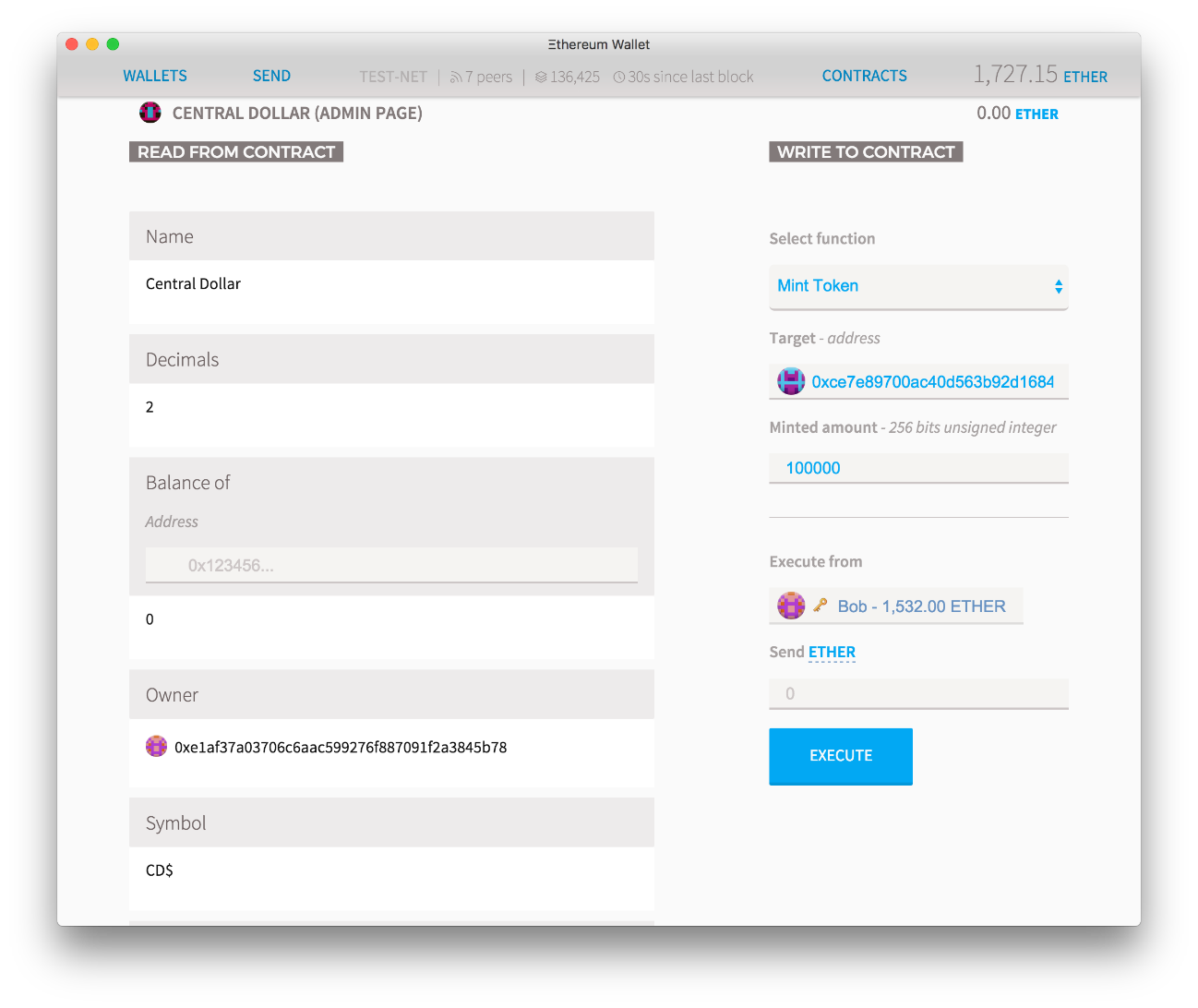
Role para baixo e você verá um custo estimado para implantação. Se você quiser, você pode alterar a barra para definir uma taxa menor, mas se o preço estiver muito abaixo da taxa média do mercado, sua transação pode demorar mais para ser processada. Clique em Deploy e digite sua senha. Depois de alguns segundos você será redirecionado para o painel e em Últimas transações (Latest transactions), você verá uma linha dizendo "creating contract". Aguarde alguns segundos para que alguém escolha sua transação e então você verá um retângulo azul lento que representa quantos outros nós viram sua transação e confirmou. Quanto mais confirmações você tiver, mais certeza você tem que seu código foi implantado.



Clique no link que diz a *página Admin* e você será direcionado ao painel de controle do banco central mais simples do mundo, onde você pode fazer o que quiser com sua moeda recém-criada.

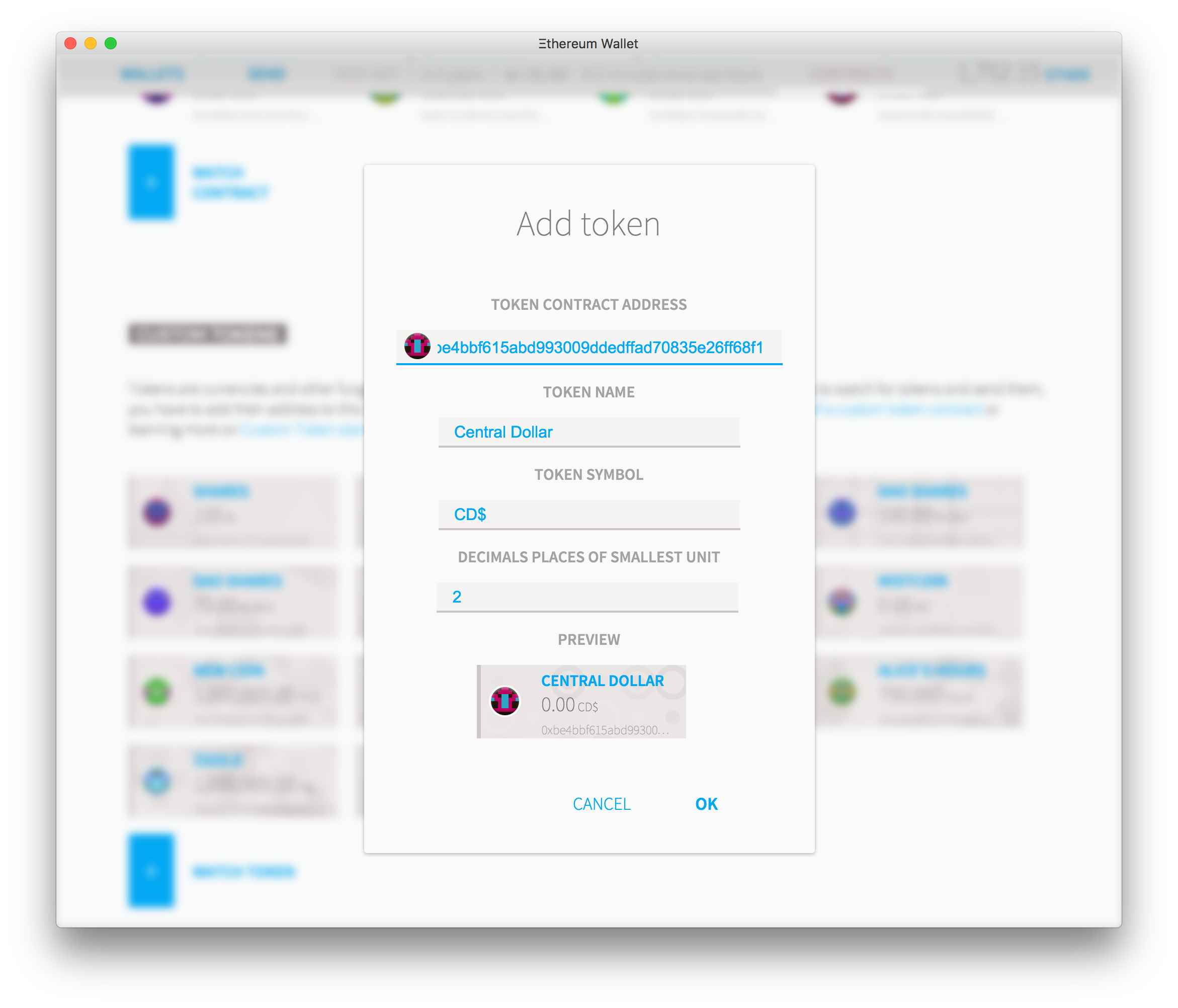
No lado esquerdo em Read from contract, você possui todas as opções e funções que você pode usar para ler as informações do contrato, gratuitamente. Se o seu token tiver um proprietário, ele exibirá seu endereço aqui. Copie esse endereço e cole-o no Saldo (Balance of) e ele irá mostrar-lhe o saldo de qualquer conta (o saldo também é exibido automaticamente em qualquer página da conta que tenha tokens).

No lado direito, sob Escrever em Contrato (Write to Contract), você verá todas as funções que você pode usar para alterar ou alterar a Blockchain de qualquer maneira. Estes irão custar gas. Se você criou um contrato que permite que você escreva moedas novas, você deve ter uma função chamada "Mint Token ". Selecione-o.



Selecione o endereço onde essas moedas novas serão criadas e, em seguida, o valor (se você tiver decimais configurado em 2, então adicione 2 zeros após o valor, para criar a quantidade correta). Em Executar (Execute from), selecione a conta que define como proprietário, deixe o montante do ether em zero e clique no botão EXECUTE.

Após algumas confirmações, o saldo do destinatário será atualizado para refletir o novo valor. Mas a carteira do destinatário pode não mostrar isso automaticamente: para ter conhecimento de tokens personalizados, a carteira deve adicioná-los manualmente a uma lista de observação. Copie o endereço do token (na página de administração, pressione copy address) e envie isso para o destinatário. Se eles ainda não o fizeram, eles deveriam ir para a aba de contratos, pressione **Watch Token** e depois adicione o endereço lá. O nome, símbolos e valores decimais exibidos podem ser personalizados pelo usuário final, especialmente se eles tiverem outros tokens com o mesmo nome (ou o mesmo). O ícone principal não é mutável e os usuários devem prestar atenção neles ao enviar e receber tokens para garantir que eles estão lidando com o negócio real e não com um token de cópia.

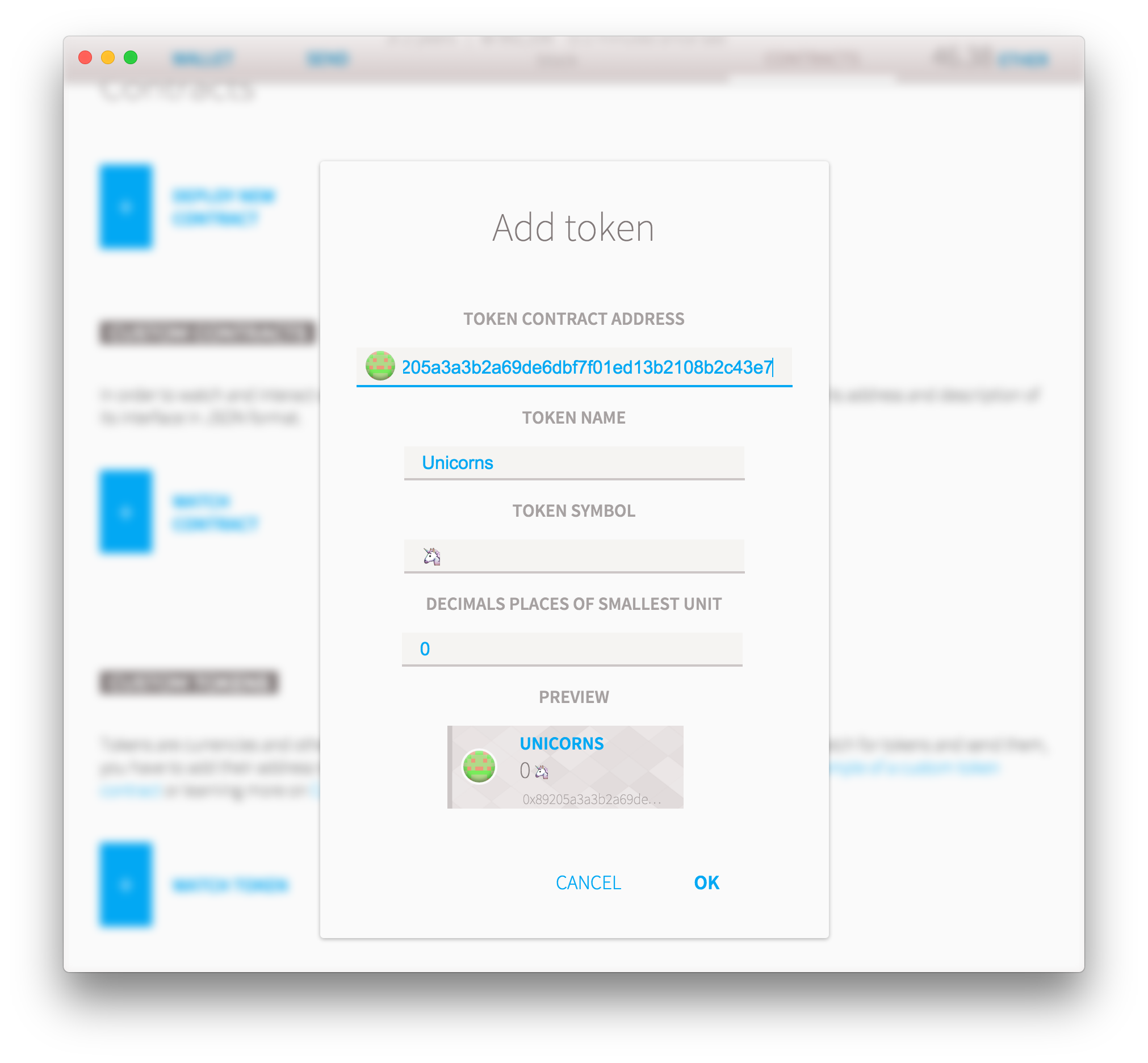


## Usando sua moeda

Uma vez que você implantou seus tokens, eles serão adicionados à sua lista de tokens e o saldo total será mostrado em sua conta. Para enviar tokens, vá para a guia Send e selecione uma conta que contenha tokens. Os tokens da conta serão listados apenas sob *Ether*. Selecione-os e digite a quantidade de tokens que deseja enviar.

Se quiser adicionar o token de outra pessoa, basta acessar a guia **Contracts** e clicar em **Watch token**.

Por exemplo, para adicionar o token do **Unicorn** à sua lista, basta adicionar o endereço **0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7** e as informações restantes serão carregadas automaticamente. Clique em *OK* e seu token será adicionado.



Os tokens Unicorn foram criados exclusivamente para aqueles que doaram para o endereço **0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359** que é controlado pela Fundação Ethereum. Para mais informações sobre eles: <https://www.ethereum.org/donate>

## E agora?

Agora que você sabe como pode usar Ethereum para emitir um token, que pode representar o que quiser. Mas o que você pode fazer com os tokens? Você pode usar, por exemplo, os tokens para representar uma participação em uma empresa ou você pode usar um comitê central para votar quando emitir novas moedas para controlar a inflação. Você também pode usá-los para arrecadar dinheiro por uma causa, através de um crowdsale.

O que você vai construir depois?