Guia Prático para Desenvolvimento de Tokens Ethereum: Introdução à Tokenização e Implementação Reprodutível

Guilherme Castro Diniz[1]
Departamento Acadêmico de Computação (DACOM)
[1]Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

The blockchain technology, particularly the Ethereum platform, has revolutionized the way assets and contracts are managed digitally. With accessible tools such as `Solidity, Remix, Ganache-cli, and Metamask`*, developers can create and validate tokens practically, expanding the possibilities of decentralized applications. The aim of this work is to create a practical guide for token development on Ethereum, covering everything from an introduction to tokenization to reproducible implementation and validation in a test environment. A smart contract was developed in* `Solidity` *using the* `Remix IDE`*, configured with version 0.8.28 and executed on the* `Cancun EVM`*. A local test network was created with* `Ganache-cli`*, allowing the validation of simulated transactions. The integration with the* `Metamask` *wallet enabled direct interaction with the token, including importing accounts and transferring values. The token was successfully implemented and tested on the local network. Transactions between simulated accounts demonstrated the contract's functionality, confirming the reproducibility of the process in a controlled environment. This practical guide provides a solid foundation for the development and validation of tokens on* `Ethereum`*. The use of modern tools like* `Remix`*,* `Ganache-cli`*, and* `Metamask` *was essential to ensuring process efficiency and reproducibility, highlighting the potential of tokenization in decentralized applications.*

Resumo

A tecnologia `blockchain`, e em particular a plataforma `Ethereum`, tem revolucionado a forma como ativos e contratos são gerenciados digitalmente. Com ferramentas acessíveis como `Solidity, Remix, Ganache-cli e Metamask`, desenvolvedores podem criar e validar tokens de maneira prática, ampliando as possibilidades de aplicações descentralizadas. O objetivo do trabalho é criar um guia prático para o desenvolvimento de tokens no `Ethereum`, abordando desde a introdução à tokenização até a implementação e validação reprodutíveis em um ambiente de teste. Foi desenvolvido um contrato inteligente em `Solidity` utilizando a `IDE Remix`, configurado com a `versão 0.8.28` e executado na `EVM Cancun`. Uma rede de testes local foi criada com o `Ganache-cli`, permitindo a validação de transações simuladas. A integração com a carteira `Metamask` possibilitou a interação direta com o token, incluindo importação de contas e transferência de valores. O token foi implementado e testado com sucesso na rede local. Transações entre contas simuladas demonstraram a funcionalidade do contrato, comprovando a reprodutibilidade do processo em um ambiente controlado. Este guia prático fornece uma base sólida para o desenvolvimento e validação de tokens no `Ethereum`. O uso de ferramentas modernas como `Remix, Ganache-cli e Metamask` foi essencial para garantir a eficiência e reprodutibilidade do processo, destacando o potencial da tokenização em aplicações descentralizadas.

Corresponding author(s): Guilherme Castro Diniz, Universidade Tecnológica Federal do Paraná (UTFPR), castrodinizg@gmail.com

1 Introdução

A tecnologia `blockchain` tem revolucionado a maneira como transações e registros são realizados no mundo digital. Introduzida inicialmente com o `Bitcoin`, essa tecnologia se baseia em uma estrutura descentralizada, imutável e transparente, capaz de armazenar dados de forma segura e confiável. Sua aplicabilidade vai muito além de moedas digitais, abrangendo áreas como finanças, saúde, logística e até mesmo o desenvolvimento de aplicações descentralizadas (DApps) (Nakamoto 2008).

Entre as diversas plataformas que utilizam a tecnologia `blockchain`, o `Ethereum` se destaca como um dos principais ecossistemas. Lançado em 2015, o Ethereum introduziu os *Smart Contracts* (Contratos Inteligentes), permitindo a execução automatizada de acordos programados diretamente na blockchain, sem a necessidade de intermediários. Esses contratos inteligentes são, essencialmente, programas de computador que garantem que as condições de um contrato sejam atendidas de forma autônoma. A ideia foi originalmente proposta no artigo de Buterin (2014) , fundador do Ethereum, e rapidamente ganhou tração como uma ferramenta poderosa para criar aplicações confiáveis e transparentes (Wood et al. 2014).

O coração do Ethereum é a Ethereum Virtual Machine (EVM), um ambiente de execução que permite que os Smart Contracts sejam processados de maneira descentralizada e segura. A EVM é responsável por garantir a compatibilidade entre os diferentes contratos e aplicações desenvolvidos na rede Ethereum, sendo fundamental para o funcionamento eficiente e interoperável do ecossistema (Wood et al. 2014) (Antonopoulos & Wood 2018).

Existem diferentes tipos de redes blockchain no Ethereum, que podem ser classificadas em redes públicas, privadas e consorciadas. Enquanto as redes públicas, como a principal rede Ethereum, são acessíveis a qualquer pessoa no mundo, as redes privadas são limitadas a um conjunto de participantes definidos, oferecendo maior controle e privacidade. Já as redes consorciadas combinam características de redes públicas e privadas, sendo utilizadas por grupos de organizações com interesses comuns (ConsenSys 2024a).

Por fim, destaca-se o conceito de tokenização, que tem transformado ativos físicos e digitais em representações digitais dentro da blockchain. Tokens podem ser utilizados para representar uma ampla gama de valores, como moedas, propriedades, ações e até mesmo direitos de uso. No Ethereum, a criação de tokens segue padrões amplamente aceitos, como o `ERC-20` para tokens fungíveis e o `ERC-721` para tokens não fungíveis (NFTs) (Foundation 2024e) (Foundation 2024c). A tokenização não apenas democratiza o acesso a ativos, mas também introduz novas possibilidades de inovação no mercado digital (Tapscott & Tapscott 2016).

Este trabalho abordará a implementação de tokens no Ethereum, explorando conceitos técnicos e práticos para permitir que outras pessoas reproduzam o processo, enquanto compreendem os fundamentos da tecnologia.

2 Fundamentação Teórica

2.1 Ethereum

O `Ethereum` é uma plataforma blockchain descentralizada que permite a criação e execução de contratos inteligentes e aplicativos descentralizados (DApps). Desde seu lançamento em 2015, a rede Ethereum tem sido amplamente adotada como base para inovações em áreas como finanças descentralizadas (DeFi) e tokenização. Dois elementos essenciais dessa tecnologia são a *Ethereum Virtual Machine* (EVM) e as diferentes redes, que incluem a rede principal e redes de teste.

A EVM é o coração do Ethereum. Trata-se de um ambiente de execução para contratos inteligentes que opera de forma descentralizada em todos os nós da rede Ethereum. Sua principal

função é garantir que o código de contratos inteligentes seja executado da mesma forma em todos os nós, proporcionando um sistema consistente e confiável (Antonopoulos & Wood 2018) (Bashir 2018).

A EVM é uma máquina virtual baseada em pilha que processa transações e implementa o consenso em um ambiente isolado. Isso significa que os contratos inteligentes não podem interagir diretamente com o sistema operacional do nó em que estão sendo executados, garantindo maior segurança. O código é escrito em linguagens como `Solidity` ou `Vyper`, que são então compiladas para *bytecode* executado pela EVM (Antonopoulos & Wood 2018) .

Características principais da EVM:

- **Turing Completa**: Permite a execução de qualquer programa computacional que possa ser logicamente concebido, dentro dos limites do "gas"disponível.
- **Isolamento**: Contratos inteligentes são executados em um ambiente virtual isolado, reduzindo riscos à rede.
- **Portabilidade**: A EVM é projetada para ser independente de hardware e sistemas operacionais, garantindo que o Ethereum possa operar em uma ampla variedade de dispositivos.

2.1.1 Redes Ethereum: Principal e de Testes

O Ethereum possui diferentes redes que servem para propósitos específicos, com a principal sendo a rede de produção e as redes de teste sendo usadas para desenvolvimento e experimentação.

A rede principal (*Mainnet*) é a rede de produção do Ethereum, onde transações reais ocorrem e têm valor econômico. É aqui que contratos inteligentes são implantados para uso público e onde os tokens emitidos têm um valor real (Antonopoulos & Wood 2018) (Foundation 2024b).

- Uso real: Todas as transações e contratos na Mainnet têm impacto econômico.
- Segurança: A Mainnet é altamente segura devido ao consenso descentralizado.
- Custo de transação: Usuários pagam "gas"(em Ether) para executar contratos ou transferir tokens.

As redes de teste são versões paralelas da blockchain Ethereum usadas para desenvolvimento, teste de aplicativos e experimentação sem o risco de gastar Ether real. Algumas das redes de teste mais populares incluem (Foundation 2024b) (Foundation 2024a):

- `Goerli`: Rede de teste cross-client que utiliza *Proof of Stake (PoS)*. Muito usada para testar atualizações do Ethereum e contratos inteligentes antes de lançá-los na Mainnet.
- `Sepolia`: Rede de teste otimizada para desenvolvimento, introduzida recentemente para substituir redes mais antigas.
- `Ropsten` (descontinuada): Era uma das redes de teste mais antigas, simulava *Proof of Work* (PoW).
- `Kovan`: Rede baseada em *Proof of Authority (PoA)*, usada para testes rápidos e sem competição.

Ether fictício: Os testnets usam Ether sem valor econômico, obtido por meio de faucets. Ambiente seguro: Desenvolvedores podem testar contratos e atualizações sem impacto na Mainnet. Propósito específico: Redes como Goerli são projetadas para testes compatíveis com múltiplos clientes.

2.2 Solidity e Contratos inteligentes

Os Contratos Inteligentes (*Smart Contracts*) são programas de computador armazenados em uma blockchain que executam automaticamente ações predefinidas quando determinadas condições

são atendidas. Esses contratos têm como principal objetivo eliminar a necessidade de intermediários, garantindo transparência e confiança nas transações digitais. O conceito foi inicialmente proposto por Szabo (1994), mas ganhou tração com o lançamento do Ethereum em 2015, que trouxe uma infraestrutura robusta para sua implementação (Buterin 2014).

Os contratos inteligentes possuem ampla aplicabilidade em diferentes setores. Eles são usados para:

- Automatizar processos financeiros: Pagamentos automáticos de empréstimos ou distribuição de royalties.
- Gerenciamento de cadeias de suprimentos: Rastreabilidade e automação de processos logísticos.
- Sistemas de votação: Garantir transparência e evitar fraudes.
- Tokenização de ativos: Representação digital de propriedades, obras de arte e outros ativos no blockchain.

Esses contratos são imutáveis e verificáveis, promovendo a confiança entre as partes envolvidas sem depender de intermediários ou de terceiros confiáveis (Buterin 2014) (Wood et al. 2014).

A rede Bitcoin foi pioneira ao introduzir a tecnologia blockchain, mas sua funcionalidade é limitada a um conjunto restrito de comandos para registrar e verificar transações. Já o Ethereum, idealizado por Vitalik Buterin, expandiu consideravelmente essas possibilidades ao introduzir a Ethereum Virtual Machine (EVM). A EVM permite executar contratos inteligentes de maneira descentralizada, criando um ecossistema onde desenvolvedores podem implementar aplicações descentralizadas (DApps) (Buterin 2014) (ConsenSys 2024b).

Com isso, o Ethereum não apenas trouxe flexibilidade na execução de contratos inteligentes, mas também impulsionou a inovação em áreas como finanças descentralizadas (DeFi), jogos e NFTs (tokens não fungíveis). Essa capacidade faz do Ethereum uma plataforma programável, em oposição à blockchain do Bitcoin, que suporta apenas scripts básicos.

Solidity é a linguagem de programação primária usada para desenvolver contratos inteligentes na rede Ethereum. Inspirada em linguagens como JavaScript, Python e C++, Solidity foi projetada para ser fácil de aprender e segura para implementar contratos complexos. Um contrato inteligente básico em Solidity geralmente inclui:

- Definição de variáveis: Para armazenar informações como saldos.
- Funções públicas e privadas: Para interagir com o contrato.
- Modificadores: Para validar permissões e condições antes da execução de funções.

O Código 1 apresente um exemplo simples de contrato inteligente em Solidity:

```
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 public storedValue;

    function set(uint256 value) public {
        storedValue = value;
    }

    function get() public view returns (uint256) {
        return storedValue;
    }
}
```

Código 1: Este contrato armazena e recupera um valor na blockchain.

Os contratos inteligentes representam uma evolução significativa na forma como transações e processos são realizados no ambiente digital. Com o suporte do Ethereum e ferramentas como Solidity, desenvolvedores têm a capacidade de criar soluções inovadoras que vão além das limi-

tações iniciais da blockchain do Bitcoin. Esse avanço tem permitido a transformação de setores inteiros, promovendo maior eficiência, transparência e acessibilidade.

2.3 Tokenização

Tokenização refere-se ao processo de representar ativos ou direitos no formato digital, utilizando a tecnologia blockchain. Tokens são entidades digitais que podem representar uma variedade de valores, desde moedas digitais e propriedades até ações e direitos de acesso. O conceito tem revolucionado diversos setores, oferecendo maior transparência, segurança e acessibilidade em transações digitais (Antonopoulos & Wood 2018) (Bashir 2018).

Um token é essencialmente uma unidade digital criada em uma blockchain para representar um ativo, utilidade ou valor. Ele pode ser fungível, como as moedas digitais (exemplo: Bitcoin ou Ether), ou não fungível (NFTs), que são únicos e usados para representar itens como obras de arte digitais ou propriedades. A emissão de tokens geralmente segue padrões pré-definidos, como os desenvolvidos no Ethereum, que garantem interoperabilidade e uniformidade (Antonopoulos & Wood 2018) (Solorio & Hoover 2019).

Os tokens podem ser classificados em diferentes categorias, dependendo de seu uso:

1. Tokens de utilidade: Oferecem acesso a serviços ou produtos dentro de um ecossistema.
2. Tokens de segurança: Representam ativos do mundo real, como ações ou títulos.
3. Tokens não fungíveis (NFTs): São únicos e amplamente utilizados para representar itens digitais ou físicos específicos.

No Ethereum, os tokens são criados usando contratos inteligentes escritos em linguagens como Solidity. Os padrões mais conhecidos incluem:

1. ERC-20: O padrão para tokens fungíveis. Ele define métodos básicos, como transferência e consulta de saldo, garantindo que os tokens sejam compatíveis com todas as aplicações que seguem este padrão (Foundation 2024e).
2. ERC-721: Padrão usado para tokens não fungíveis (NFTs). Cada token é único e pode ter metadados específicos associados a ele (Foundation 2024c).
3. ERC-1155: Um padrão híbrido que permite criar tokens fungíveis e não fungíveis em um único contrato inteligente, otimizando custos e complexidade (Foundation 2024c).

Esses padrões fornecem uma base sólida para a criação de tokens, promovendo a interoperabilidade em ecossistemas blockchain.

Tokens funcionam como contratos inteligentes que definem as regras de interação. Por exemplo, um token ERC-20 possui funções como transfer (para transferir tokens entre contas) e balanceOf (para verificar o saldo). Esses contratos inteligentes são executados de forma descentralizada na blockchain, eliminando a necessidade de intermediários.

Os tokens têm ampla utilidade em vários setores:

1. Finanças descentralizadas (DeFi): Utilizados para empréstimos, poupança e outros serviços financeiros sem intermediários.
2. Economia digital: Pagamentos, recompensas em jogos e sistemas de fidelidade.
3. NFTs: Representação de itens únicos, como arte digital, imóveis ou colecionáveis.
4. Governança: Tokens podem dar direito a voto em decisões de plataformas descentralizadas (Bashir 2018).

A tokenização democratiza o acesso a ativos, permitindo que qualquer pessoa participe de mercados antes restritos a grandes instituições.

3 Materiais e métodos

Nesta seção, são descritas as ferramentas utilizadas para o desenvolvimento e implementação de contratos inteligentes, bem como o processo de instalação e configuração em um sistema operacional Linux/Ubuntu.

3.1 Ferramentas Utilizadas no Desenvolvimento

3.1.1 Linguagem e IDE

O Solidity é a principal linguagem de programação utilizada para desenvolver contratos inteligentes na plataforma Ethereum. Inspirada em linguagens como JavaScript, C++ e Python, Solidity permite a criação de contratos seguros e confiáveis que são executados na Ethereum Virtual Machine (EVM). Com ela, desenvolvedores podem implementar funcionalidades complexas, como sistemas de votação, carteiras digitais e marketplaces

O Remix é uma IDE (Ambiente de Desenvolvimento Integrado) baseada na web que facilita a criação, compilação e implantação de contratos inteligentes. Ele fornece um ambiente completo, incluindo um compilador de Solidity, um simulador de blockchain e ferramentas para depuração. Por ser uma plataforma online, o Remix não requer instalação e pode ser acessado diretamente pelo navegador no link Remix IDE[1] (Foundation 2024d).

Utilizei o Remix para desenvolver o código em Solidity, realizar a compilação e criar o contrato inteligente. Essa ferramenta foi essencial para testar e verificar o comportamento do contrato antes de sua implementação em um ambiente blockchain. O processo foi realizado diretamente na interface do Remix, permitindo depuração em tempo real e validação eficiente do código.

3.1.2 Simulador de Blockchain - Ganache-cli

O Ganache-cli é uma ferramenta poderosa para o desenvolvimento e teste de contratos inteligentes. Trata-se de uma blockchain local que simula o comportamento da Mainnet ou de redes de teste do Ethereum, permitindo o teste rápido de transações e contratos sem custos reais. Ele é amplamente utilizado em conjunto com bibliotecas como Truffle e Web3.js (Antonopoulos & Wood 2018).

Instalação do Ganache-cli no Linux/Ubuntu. Certifique-se de ter o Node.js e o npm instalados. Em seguida, instale o Ganache-cli globalmente:

```
1 npm install ganache --global
```

Código 2: Instalação do Ganache-cli globalmente

```
1 ganache-cli --version
```

Código 3: Verifique a versão instalada para confirmar a instalação.

3.1.3 Carteira Metamask

A Metamask é uma carteira digital (*wallet*) que permite gerenciar criptomoedas e tokens baseados no Ethereum, interagir com contratos inteligentes e conectar-se a diferentes redes blockchain. Disponível como uma extensão para navegadores e aplicativo móvel, a Metamask é amplamente

[1] https://remix.ethereum.org/

utilizada por desenvolvedores e usuários para interagir com redes Ethereum, como a Mainnet e redes de teste, incluindo aquelas criadas localmente com ferramentas como Ganache-cli.

Suporte para envio, recebimento e gerenciamento de tokens baseados em padrões como ERC-20 e ERC-721. Conexão direta com dApps (aplicações descentralizadas) e IDEs como o Remix. Interface simples para adicionar e gerenciar redes personalizadas, como uma rede local configurada com Ganache-cli. Recurso de importação de tokens, permitindo que os usuários adicionem contratos personalizados à carteira.

3.2 Metodologia de Desenvolvimento

3.2.1 Desenvolvimento do Contrato Inteligente

Para a criação do contrato, foi utilizada a IDE do Remix. Essa ferramenta permitiu o desenvolvimento e a compilação do código em Solidity, além de possibilitar a simulação da execução do contrato em um ambiente controlado. Para a compilação, foi utilizada a versão Solidity 0.8.28, configurada diretamente no compilador da IDE. A execução foi simulada na EVM Cancun, uma das versões mais recentes da Ethereum Virtual Machine, garantindo compatibilidade com os padrões atuais da rede Ethereum.

O código desenvolvido pode ser avaliado abaixo:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.28;
3
4  contract TokenMinas {
5      string public name = "Token Minas";
6      string public symbol = "TKM";
7
8      uint8 public decimals = 0;
9      uint256 private _totalSupply = 100;
10
11     mapping(address => uint256) public _balances;
12     mapping(address => mapping(address => uint256)) public _allowed;
13
14     event Transfer(address indexed from, address indexed to, uint256 value);
15     event Approval(
16         address indexed owner,
17         address indexed spender,
18         uint256 value
19     );
20
21     constructor() {
22         _balances[msg.sender] = _totalSupply;
23         emit Transfer(address(0), msg.sender, _totalSupply);
24     }
25
26     function totalSupply() public view returns (uint) {
27         return _totalSupply - _balances[address(0)];
28     }
29
30     function balanceOf(address _owner) public view returns (uint balance) {
31         return _balances[_owner];
32     }
33
34     function allowance(address _owner, address _spender) public view returns (
           uint remaining) {
35         return _allowed[_owner][_spender];
36     }
```

```
37
38      function transfer(address to, uint256 value) public returns (bool success) {
39          require(_balances[msg.sender] >= value, "Saldo insuficiente");
40          _balances[msg.sender] -= value;
41          _balances[to] += value;
42          emit Transfer(msg.sender, to, value);
43          return true;
44      }
45
46      function approve(
47          address spender,
48          uint256 value
49      ) public returns (bool success) {
50          _allowed[msg.sender][spender] = value;
51          emit Approval(msg.sender, spender, value);
52          return true;
53      }
54
55      function transferFrom(
56          address from,
57          address to,
58          uint256 value
59      ) public returns (bool success) {
60          require(_balances[from] >= value, "Saldo insuficiente");
61          require(
62              _allowed[from][msg.sender] >= value,
63              "Sem permissao suficiente"
64          );
65
66          _balances[from] -= value;
67          _balances[to] += value;
68          _allowed[from][msg.sender] -= value;
69
70          emit Transfer(from, to, value);
71          return true;
72      }
73  }
```

Código 4: Código do contrato

3.2.2 Configuração da Rede de Testes

Após o desenvolvimento do contrato, foi configurada uma rede de testes local no computador para verificar sua funcionalidade. Para isso, foi utilizado o Ganache-cli, uma ferramenta que permite criar uma blockchain local para simulação.

Para iniciar a rede de testes, abra o terminal do Linux e digite o seguinte comando:

```
1       $ ganache-cli
```

Código 5: Criar rede de teste local

Se todas as dependências estiverem instaladas corretamente, o comando irá instanciar uma rede local no endereço **127.0.0.1:8545**. Essa rede será composta por 10 contas, cada uma com um saldo inicial de 1000 ETH, e suas respectivas private keys serão exibidas.

O resultado esperado no terminal deve ser semelhante ao da figura 1:

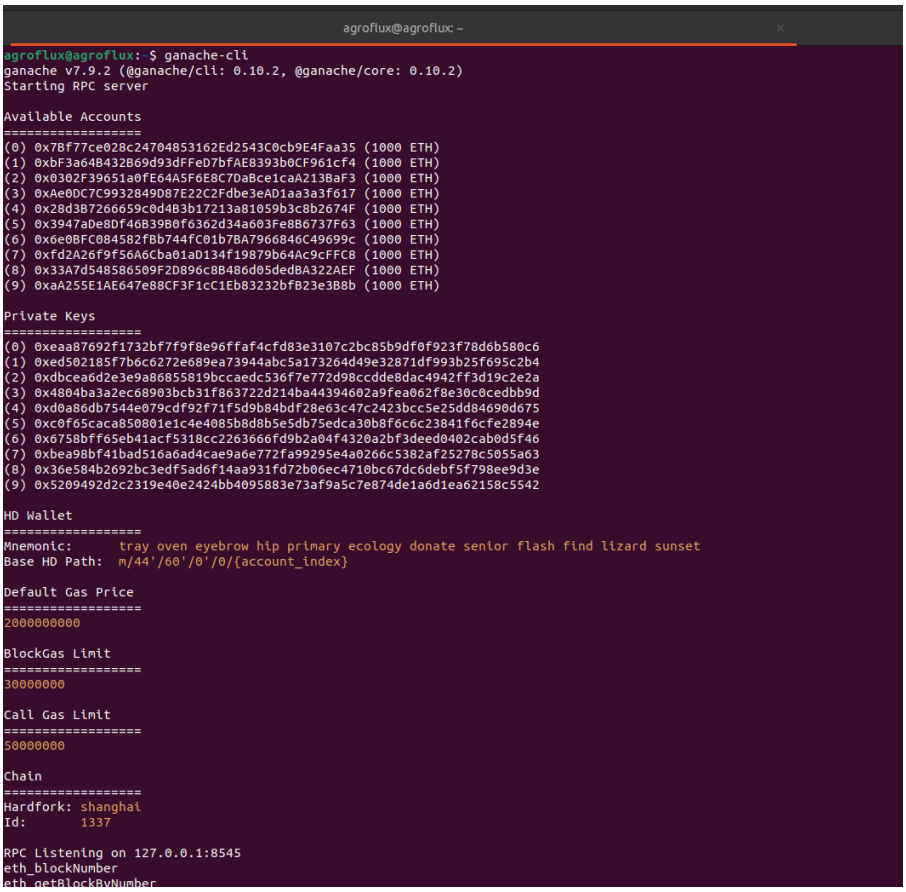

Figura 1: Criação da rede de teste local utilizando o ganache-cli.

Essa configuração permite que o contrato inteligente seja implantado e testado de maneira local, simulando as condições de uma rede Ethereum real, mas sem custos associados.

3.2.3 Conexão do Remix à Rede de Testes Local

Após criar o contrato inteligente, compilá-lo e configurar a rede de testes local utilizando o Ganache-cli, o próximo passo é conectar o Remix à rede local para prosseguir com o deploy do contrato.

Conectando o Remix à Rede Local:

1. Acesse a aba ”Deploy and run transactions” na interface do Remix.
2. Na seção ”*Environment*”, selecione a opção ”Dev - Ganache Provider”. Isso indica ao Remix que ele deve utilizar o provedor do Ganache-cli para interagir com a rede de testes local.
3. Confirme a conexão com a rede local. Se a conexão for bem-sucedida, as contas criadas pelo Ganache-cli (com saldo de 1000 ETH cada) serão exibidas na seção ”ACCOUNT” da interface do Remix.

O resultado esperado é semalhante ao da figura 2:

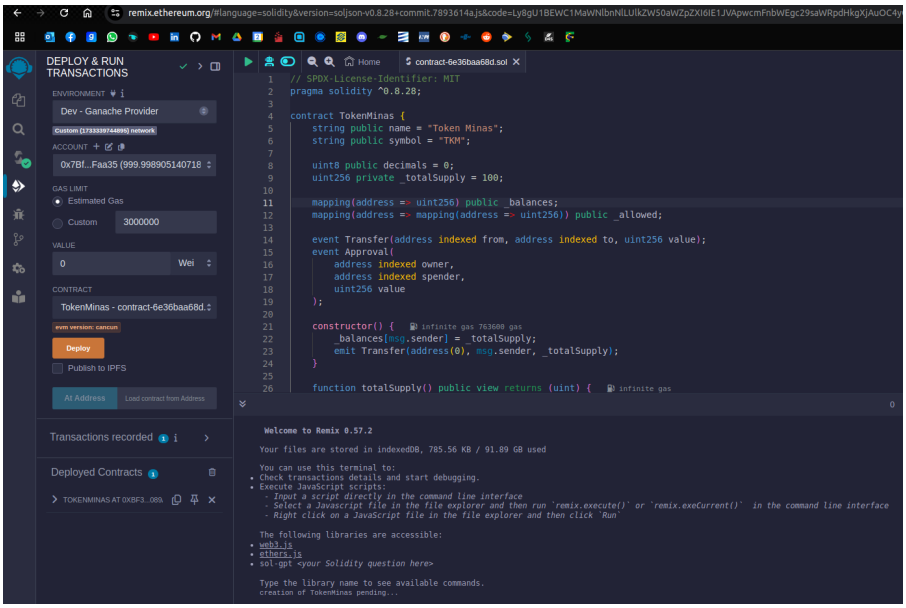

Figura 2: Conexão do Remix com a rede local Ganache Cli

3.2.4 Realizando o *Deploy* do Contrato

1. Certifique-se de que o contrato foi selecionado corretamente na seção ”Contract”do Remix. Clique no botão ”Deploy”.
2. Este comando enviará o contrato inteligente para a rede de testes local gerenciada pelo Ganache-cli.
3. No terminal do Remix, você verá um feedback detalhado sobre o contrato implantado, incluindo o endereço do contrato na blockchain local e outras informações relevantes.
4. O resultado no Remix e também na rede ganache-cli local será como o da figura 3 e 4.

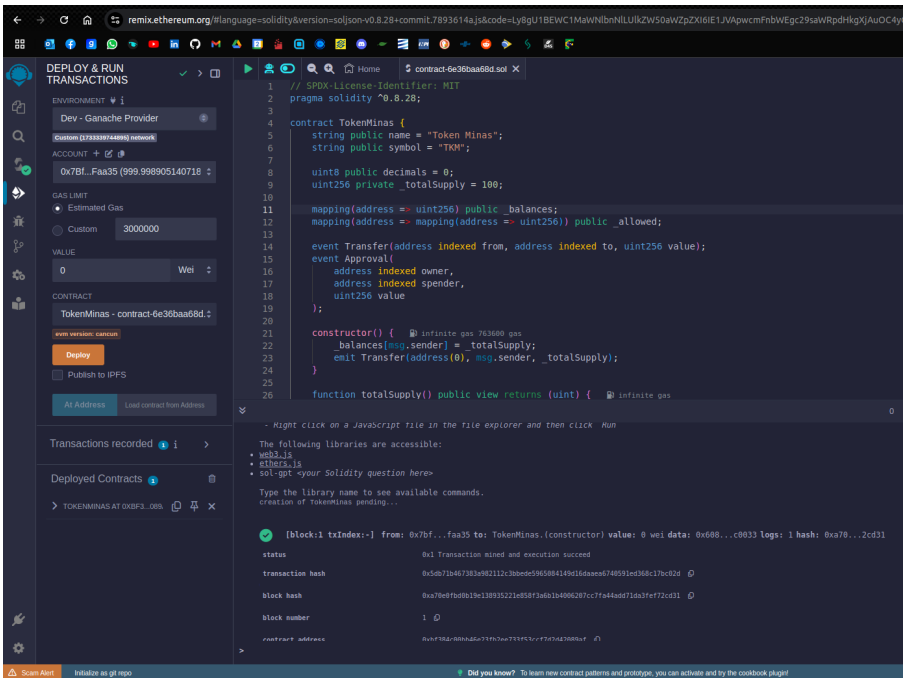

Figura 3: Deploy realizado com sucesso. Imagem da IDE Remix

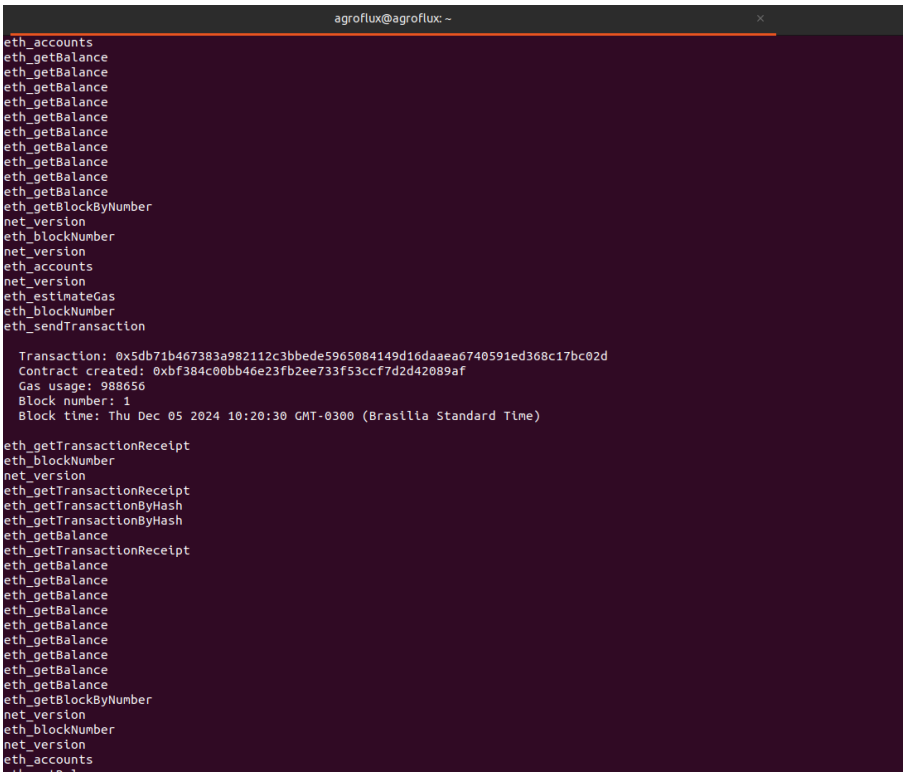

Figura 4: Deploy realizado com sucesso. Imagem da rede local ganache-cli.

3.2.5 Configuração da Rede Local no Metamask

Após conectar o Remix à rede local e realizar o deploy do contrato inteligente, foi utilizada a Metamask para testar a interação com o token criado. A Metamask permitiu realizar transferências simuladas e validar a funcionalidade do contrato na rede local configurada pelo Ganache-cli. É importante que você criei uma conta e tenha a sua carteira. Após criação da conta, vamos conectar o Metamask no rede local ganache-cli.

1. Abra a Metamask e acesse a aba ”Configurações”.
2. Selecione ”Redes”e clique em ”Adicionar uma rede”.
3. Insira os detalhes da rede local configurada pelo Ganache-cli:
   - Nome da rede: Ganache Local
   - Novo URL RPC: http://127.0.0.1:8545
   - ID da cadeia: 1337 (ou o ID configurado pelo Ganache)
   - Símbolo da moeda: ETH
   - URL do explorador de blocos: (deixe vazio, se preferir)
4. Clique em ”Salvar”. A rede local será adicionada e poderá ser selecionada na interface principal da Metamask.

3.2.6 Importação de Contas do Ganache-cli

Para realizar as transações de teste, foi necessário importar no mínimo duas contas geradas pelo Ganache-cli para dentro da Metamask. Cada conta criada pelo Ganache possui uma private key, exibida no terminal ao iniciar o Ganache-cli.

1. No terminal do Ganache-cli, copie a private key de uma conta (por exemplo, (0) Private Key: 0xabc123...).
2. Na Metamask, clique em ”Importar Conta”.
3. Cole a private key da conta no campo correspondente e clique em ”Importar”.

4. Repita o processo para uma segunda conta. Ambas as contas devem estar disponíveis na interface da Metamask, com saldo inicial de 1000 ETH cada, conforme configurado pelo Ganache-cli.
5. Sua conta importada deverá ficar semelhante a figura 5.

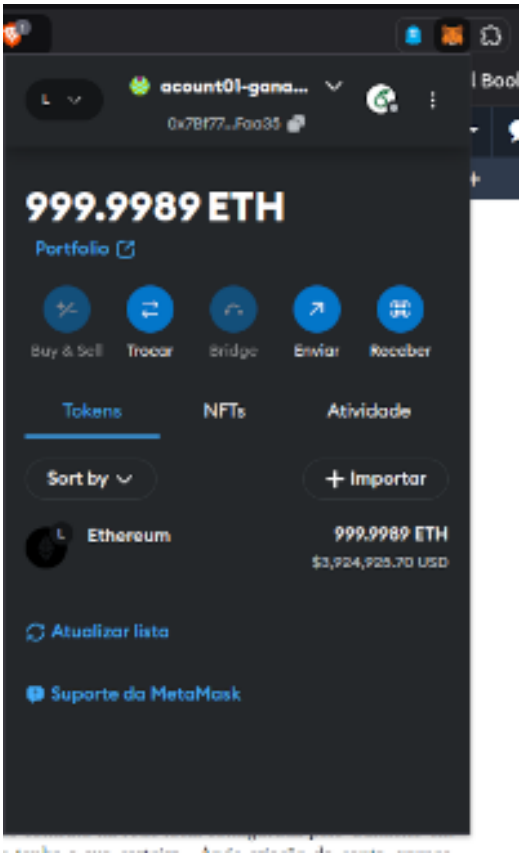

Figura 5: Conta da rede local Ganache-cli conectada a carteira metamask.

3.2.7 Importação do Token no Metamask

1. Após realizar o deploy do contrato no Remix, copie o endereço do contrato gerado. Você poderá encontrar o endereço do contrato no terminal do ganache-cli, com o nome ”Contract created” , conforme figura 4
2. Na Metamask, clique em ”Importar Tokens”e insira o endereço do contrato no campo ”Endereço do Token”e o simbolo do Token definido no contrato, no meu caso TKM. A imagem 8 mostra como os campos devem ser preenchidos.
3. Confirme a importação clicando em ”Seguinte”.
4. Você deverá realizar o processo de importar o Token para as duas contas importadas da rede Ganache-cli.

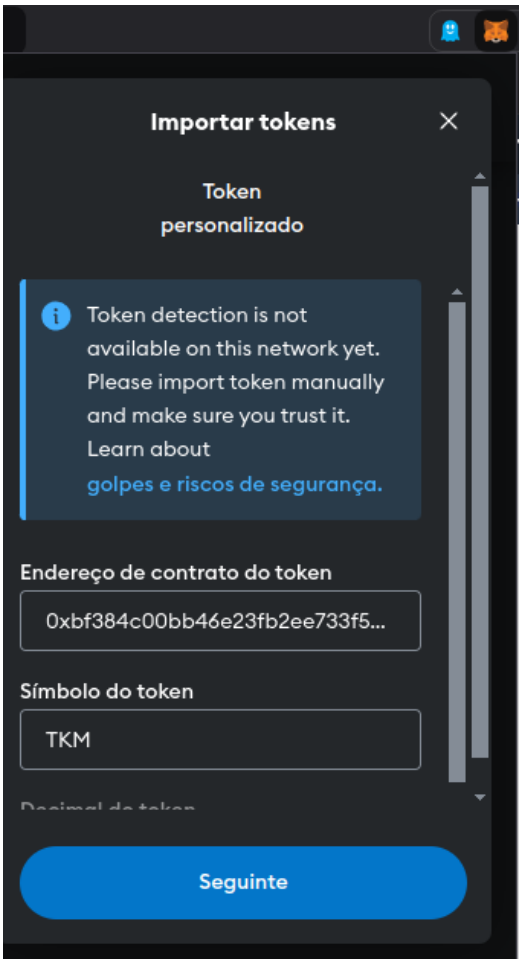

Figura 6: Importando Token a contas do Ganache-cli

3.2.8 Testes de Transferência

1. No MetaMask em uma das contas (geraknebte na primeira que o token foi importada) você deverá ter um Saldo, que foi definida quando o contrato foi criado. No meu caso foi o valor 100 TKM.
2. Clique em "Enviar", insira o endereço da outra conta (também importada do Ganache), selecione o Token importado (TKM) e depois defina o valor do token a ser transferido.
3. Confirme a transação. A transferência será processada na rede local simulada pelo Ganache-cli. Na figura 7 e 8 é possível avaliar se a transação ocorreu com sucesso.
4. Após a transação, verifique o saldo atualizado em ambas as contas na interface da Metamask. Na figura 9é mostrado o saldo da conta de destino, ou seja, da conta que recebeu a transaferência.

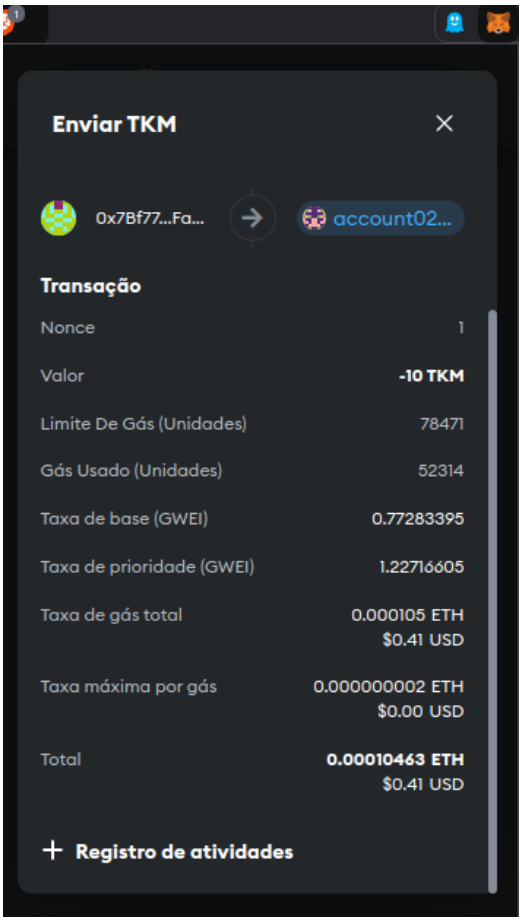

Figura 7: Detalhes da transação de envio de 10TKM entre contas do ganache-cli.

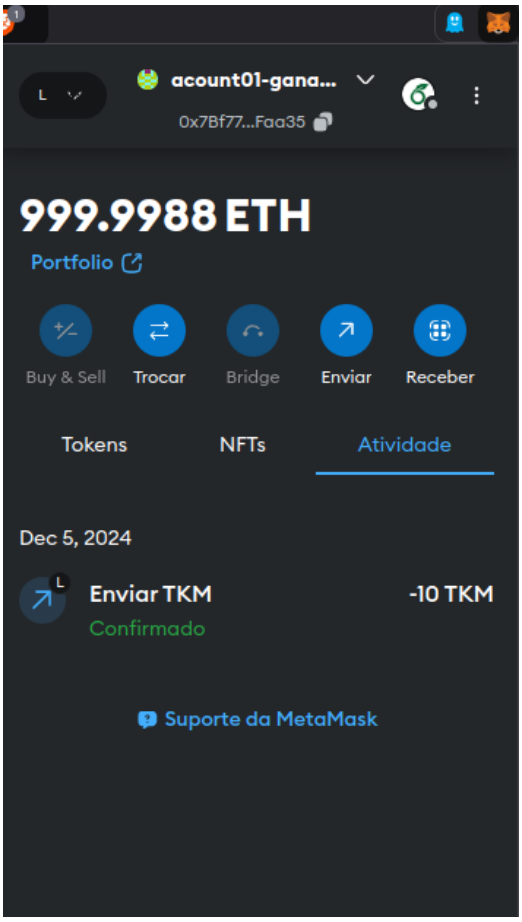

Figura 8: Em ”Atividades” é possível analisar a confirmação do envio dos tokens.

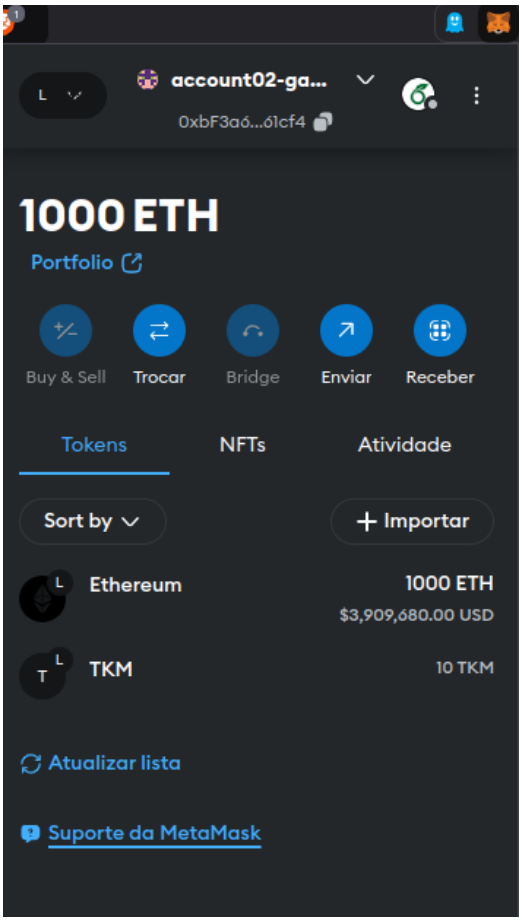

Figura 9: Conta de destino que recebeu os 10 TKM. É possível analisar pelo saldo que a transação foi realizada com sucesso.

4 Conclusão

O presente trabalho detalhou o processo de desenvolvimento e implantação de um contrato inteligente utilizando as ferramentas mais relevantes no ecossistema Ethereum. A escolha de tecnologias como Solidity, Remix, Ganache-cli e Metamask demonstrou ser eficiente e prática, possibilitando desde a criação do código até a validação em uma rede local simulada.

A utilização do Remix permitiu um ambiente integrado para a escrita, compilação e deploy do contrato inteligente, enquanto o Ganache-cli proporcionou uma rede de testes local, essencial para garantir um ambiente seguro e controlado. A integração com a Metamask demonstrou como a interação com contratos e tokens pode ser realizada de maneira simples e intuitiva, mesmo em uma rede personalizada.

Com essas ferramentas, foi possível realizar o deploy do contrato e validar sua funcionalidade por meio de transações simuladas, destacando a capacidade de reproduzir cenários reais de uso. Essa abordagem garante não apenas a confiabilidade do código desenvolvido, mas também a preparação para sua eventual implantação em redes públicas ou privadas do Ethereum.

Esse trabalho reforça a importância de ferramentas modernas e acessíveis no desenvolvimento blockchain e evidencia como elas podem ser utilizadas para explorar o potencial de tecnologias descentralizadas, garantindo inovação e confiabilidade no contexto de contratos inteligentes.

Referências

Antonopoulos, Andreas M. & Gavin Wood. 2018. *Mastering ethereum: Building smart contracts and dapps*. O'Reilly Media. https://books.google.com.br/books?id=SedSMQAACAAJ.

Bashir, Imran. 2018. *Mastering blockchain: Distributed ledger technology, decentralization, and smart contracts explained, 2nd edition*. Packt Publishing. https://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=1789486&lang=pt-br&site=eds-live&scope=site.

Buterin, Vitalik. 2014. Ethereum white paper: A next-generation smart contract and decentralized application platform. Accessed: 2024-12-02. https://ethereum.org/en/whitepaper/.

ConsenSys. 2024a. Blockchain networks: Public vs. private vs. consortium. Accessed: 2024-12-02. https://consensys.net/knowledge-base/blockchain/.

ConsenSys. 2024b. Introduction to solidity programming. Accessed: 2024-12-02. https://docs.soliditylang.org/.

Foundation, Ethereum. 2024a. Goerli testnet guide. https://goerli.net/.

Foundation, Ethereum. 2024b. Introduction to ethereum networks. https://ethereum.org/en/developers/docs/networks/.

Foundation, Ethereum. 2024c. Introduction to token standards. https://ethereum.org/en/developers/docs/standards/tokens/.

Foundation, Ethereum. 2024d. Remix ide documentation. https://remix.ethereum.org/.

Foundation, Ethereum. 2024e. Standards - erc-20 and erc-721. https://eips.ethereum.org/.

Nakamoto, Satoshi. 2008. Bitcoin: A peer-to-peer electronic cash system. Accessed: 2024-12-02. https://bitcoin.org/bitcoin.pdf.

Solorio, Kanna R. Hoover D., K. & D. H. Hoover. 2019. *Hands-on smart contract development with solidity and ethereum: From fundamentals to deployment*. O'Reilly Media. https://books.google.com.br/books?id=vYRkwgEACAAJ.

Szabo, Nick. 1994. Smart contracts: Building blocks for digital markets. Accessed: 2024-12-02. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

Tapscott, Don & Alex Tapscott. 2016. *Blockchain revolution: How the technology behind bitcoin is changing money, business, and the world*. Penguin.

Wood, Gavin et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151(2014). 1–32.