

# Implementando tokens no Ethereum

Geovana Figueiredo Silva<sup>1</sup>

<sup>1</sup>Departamento Acadêmico de Computação (DACOM)  
Universidade Tecnológica Federal do Paraná (UTFPR)

## Abstract

This work addresses the implementation of *tokens* on the *Ethereum blockchain*, focusing on the practical application of the ERC-20 standard for creating digital assets. The study delves into fundamental concepts such as the *Ethereum* Virtual Machine (EVM), smart contracts, and tokenization, leveraging tools like *Ganache* and *Truffle* for the development and testing of smart contracts. Through the implementation of a fungible token, the research demonstrates how well-established standards can streamline the creation and management of *tokens* on the *blockchain*. The results highlight the technology's potential to transform various sectors, such as finance and governance, while discussing challenges and opportunities for future advancements.

## Resumo

Este trabalho aborda a implementação de *tokens* na *blockchain Ethereum*, destacando a aplicação prática do padrão ERC-20 para a criação de ativos digitais. A pesquisa detalha os conceitos fundamentais da *Ethereum* Virtual Machine (EVM), contratos inteligentes e tokenização, além de utilizar ferramentas como *Ganache* e *Truffle* para o desenvolvimento e teste de contratos inteligentes. Com a implementação de um token fungível, o estudo demonstra como padrões bem estabelecidos podem simplificar o processo de criação e gestão de *tokens* na *blockchain*. Os resultados evidenciam o potencial da tecnologia para transformar setores diversos, como finanças e governança, ao mesmo tempo em que discutem desafios e oportunidades para melhorias futuras.

## 1 Introdução

As tecnologias *blockchain* têm revolucionado a maneira como interagimos com sistemas digitais, oferecendo uma infraestrutura descentralizada, transparente e segura para a realização de transações e execução de contratos. Inicialmente associada ao *Bitcoin*, a *blockchain* evoluiu para muito além de sua aplicação como base para criptomoedas, encontrando usos em diversas áreas como finanças, cadeia de suprimentos, saúde e governança.

Entre as plataformas que exploram todo o potencial da *blockchain*, o *Ethereum* se destaca por seu design inovador, que permite a criação de contratos inteligentes (smart contracts) e aplicativos descentralizados (dApps). Diferentemente do *Bitcoin*, o *Ethereum* não se limita a transações financeiras, sendo projetado para ser uma plataforma global de execução de código programável.

No contexto do *Ethereum*, a tokenização é uma aplicação que vem ganhando destaque. *tokens* são representações digitais de ativos, direitos ou valores, que podem ser criados e gerenciados na *blockchain*. Esses *tokens* desempenham um papel fundamental em economias digitais, sendo

utilizados para representar desde moedas e pontos de fidelidade até ações de empresas e propriedades virtuais.

Este trabalho explora o processo de implementação de *tokens* no *Ethereum*, abordando desde os fundamentos da tokenização até as práticas para o desenvolvimento de contratos inteligentes que gerenciem esses ativos digitais. Ao longo do texto, será discutido como os *tokens* podem ser aplicados para transformar modelos econômicos tradicionais, impulsionando inovações em diversos setores.

## 2 Ethereum

O *Ethereum* é uma plataforma descentralizada que permite a criação e execução de contratos inteligentes e aplicativos descentralizados. Sua arquitetura foi projetada para ser altamente flexível, permitindo que desenvolvedores programem funcionalidades personalizadas em sua *blockchain*.

### 2.1 Ethereum Virtual Machine (EVM)

No coração do *Ethereum* está a *Ethereum Virtual Machine* (EVM), uma máquina virtual descentralizada que executa contratos inteligentes. A EVM é responsável por interpretar e executar o bytecode dos contratos, garantindo que todas as operações sejam realizadas de maneira consistente em todos os nós da rede.

A EVM é uma máquina **Turing-completa**, o que significa que pode executar qualquer cálculo computacional dado tempo e recursos suficientes. Sua arquitetura foi projetada para ser determinística e segura, evitando resultados inconsistentes entre os nós. Cada instrução executada na EVM consome gas, uma unidade de medida que limita os recursos computacionais utilizados por um contrato, garantindo que a rede permaneça eficiente e livre de loops infinitos.

Os contratos inteligentes na EVM são escritos em linguagens de alto nível, como *Solidity* ou *Vyper*, e depois compilados para bytecode antes de serem implantados na *blockchain*. Este modelo garante que os contratos sejam imutáveis e públicos, promovendo transparência e confiança entre os usuários.

### 2.2 Redes Ethereum: Principal e de Testes

O *Ethereum* opera em várias redes, cada uma com propósitos distintos. Essas redes permitem que desenvolvedores e usuários realizem experimentos e transações com segurança antes de interagir com a *blockchain* principal.

#### 2.2.1 Rede Principal

A rede principal, ou *mainnet*, é a *blockchain* pública onde transações reais são executadas e possuem valor econômico. É nesta rede que os contratos inteligentes são implantados para uso em larga escala, e onde os *tokens* desenvolvidos podem ser trocados por *Ether* (ETH) ou outros ativos digitais.

Devido à sua natureza pública e descentralizada, a *mainnet* é altamente segura, mas também possui custos associados, como taxas de gas, que podem variar de acordo com a demanda da rede.

### 2.2.2 Redes de Teste

As redes de teste, ou *testnets*, são *blockchains* paralelas à rede principal, projetadas para simular o ambiente real sem os custos associados à *mainnet*. São usadas por desenvolvedores para testar contratos inteligentes, aplicativos e interações antes de implantá-los de forma definitiva.

Entre as principais *testnets* do *Ethereum*, destacam-se:

- **Goerli:** Uma das *testnets* mais populares, utilizada para testes de contratos inteligentes e aplicativos descentralizados.
- **Sepolia:** Uma rede de teste eficiente e amplamente suportada.
- **Ropsten (descontinuada):** Era conhecida por simular condições da *mainnet*, mas foi substituída por outras redes mais modernas.

Nas *testnets*, o *Ether* não possui valor econômico, podendo ser obtido gratuitamente em *faucets* (distribuidores de ETH de teste). Isso torna as *testnets* ambientes ideais para experimentação e aprendizado sem riscos financeiros.

Ao trabalhar com o *Ethereum*, compreender a EVM e as diferentes redes disponíveis é essencial para o desenvolvimento de soluções robustas e seguras. Esses aspectos serão fundamentais para a implementação de *tokens* descrita nos próximos capítulos deste trabalho.

## 3 Contratos inteligentes

Os contratos inteligentes, conceito introduzido por Nick Szabo em 1997, são protocolos digitais que executam automaticamente os termos de um contrato sem a necessidade de intermediários humanos. De acordo com Szabo, esses contratos são projetados para minimizar a necessidade de confiança, utilizando tecnologia para garantir o cumprimento dos acordos estabelecidos Antonopoulos & Wood (2018).

No contexto das *blockchains*, contratos inteligentes são programas armazenados em redes descentralizadas que executam ações predeterminadas quando condições específicas são atendidas. Por exemplo, podem transferir *tokens* de uma conta para outra após a validação de uma transação ou criar um registro imutável de propriedade.

### 3.1 Usos e Utilidade

Os contratos inteligentes têm aplicações em diversos setores:

- **Financeiro:** Facilitação de pagamentos automáticos, criação de *tokens* e negociação de ativos digitais, incluindo contratos complexos no mercado de derivativos.
- **Imobiliário:** Automação de transferências de propriedade após pagamentos, reduzindo custos e eliminando burocracias.
- **Logística e Cadeias de Suprimentos:** Registro de informações sobre o transporte de mercadorias, garantindo transparência e rastreabilidade.
- **Identidade e Certificações:** Verificação e registro de credenciais educacionais ou profissionais de maneira imutável e auditável.

### 3.2 Benefícios

- **Transparência:** As regras do contrato são públicas e auditáveis, criando confiança.
- **Eficiência:** Redução de custos operacionais ao eliminar intermediários.
- **Imutabilidade:** Os registros são resistentes a alterações ou fraudes após serem gravados na *blockchain*.
- **Segurança:** Utilizam protocolos criptográficos para proteger a integridade dos dados.

### 3.3 Contratos Inteligentes no *Ethereum*

A *blockchain Ethereum* é amplamente reconhecida como a plataforma pioneira para a implementação de contratos inteligentes. Por meio de sua linguagem de programação *Solidity*, desenvolvedores podem criar contratos que automatizam funções de negócios ou serviços.

Um exemplo comum no *Ethereum* é a criação de *tokens* ERC-20, que obedecem a padrões que garantem interoperabilidade. Esses contratos são amplamente utilizados em ofertas iniciais de moedas (ICOs), no mercado de NFTs e em projetos DeFi (finanças descentralizadas) Antonopoulos & Wood (2018).

Os contratos inteligentes representam um marco na evolução tecnológica, com o potencial de remodelar a forma como transações e interações digitais são realizadas, promovendo um ecossistema mais confiável e eficiente.

## 4 *Solidity* e Implementação de Contratos Inteligentes

### 4.1 Introdução à *Solidity*

*Solidity* é a principal linguagem de programação utilizada para o desenvolvimento de contratos inteligentes na *blockchain Ethereum*. Criada especificamente para esta finalidade, *Solidity* é uma linguagem orientada a objetos, inspirada por outras linguagens como *JavaScript*, *Python* e C++.

Com uma sintaxe clara e recursos robustos, *Solidity* permite que desenvolvedores escrevam contratos inteligentes que sejam:

- **Determinísticos:** O mesmo código produzirá os mesmos resultados em qualquer nó da rede *Ethereum*.
- **Seguros:** Suas funcionalidades incluem controle de acesso, validações e prevenção contra comportamentos inesperados.
- **Efetivos:** Projetada para gerenciar transferências de ativos e automatizar transações, reduzindo custos operacionais e aumentando a eficiência.

### 4.2 Principais Componentes da *Solidity*

#### 1. Estrutura do Contrato

Um contrato em *Solidity* é um conjunto de códigos e dados armazenados na *blockchain*. Ele pode conter:

- **Funções:** Para executar ações específicas.
- **Variáveis de Estado:** Para armazenar informações persistentes no *blockchain*.
- **Eventos:** Para registrar dados que podem ser consultados fora do *blockchain*.

Exemplo básico no trecho de código 1:

```
1  contract MeuContrato {
2      string public mensagem;
3
4      constructor(string memory _mensagem) {
5          mensagem = _mensagem;
6      }
7
8      function atualizarMensagem(string memory _novaMensagem) public {
9          mensagem = _novaMensagem;
10     }
11 }
```

Código 1: Exemplo de código

Este exemplo ilustra um contrato simples que armazena e atualiza uma mensagem na *blockchain*.

## 2. Segurança em *Solidity*:

linguagem foi desenvolvida com foco em segurança, mas como em qualquer desenvolvimento de software, erros podem ocorrer. Para reduzir riscos:

- Utilize bibliotecas confiáveis, como a *OpenZeppelin*, para funções padrão.
- Implemente verificações de segurança, como limites de acesso com o modificador *‘onlyOwner’*.
- Teste rigorosamente o contrato em ambientes de teste antes de publicá-lo.

## 4.3 Implementação de Contratos Inteligentes

A implementação de contratos inteligentes no *Ethereum* envolve as seguintes etapas:

1. **Escrita do Código** Escreva o contrato inteligente usando um ambiente de desenvolvimento integrado (IDE), como o [Remix](https://remix.Ethereum.org/). Este IDE baseado na web é amplamente utilizado para compilar e testar contratos *Solidity*.
2. **Compilação** O código escrito em *Solidity* é compilado em bytecode, uma representação que pode ser executada pela *Ethereum Virtual Machine* (EVM).
3. **Implantação na blockchain** O contrato é implantado na *blockchain Ethereum* através de ferramentas como *Truffle*. A implantação requer uma conta *Ethereum* e o pagamento de "gas fees" (taxas de execução).
4. **Interação com o Contrato** Após a implantação, os contratos podem ser acessados por qualquer pessoa na *blockchain* utilizando aplicativos descentralizados (DApps) ou bibliotecas como *Web3.js*.

## 4.4 Exemplo Prático: Criando um Token ERC-20

O padrão ERC-20 é amplamente utilizado para a criação de *tokens* fungíveis no *Ethereum*. Aqui está um exemplo de implementação básica:

```
1 import "@textbf{\textit{OpenZeppelin}}/contracts/token/ERC20/ERC20.sol";
2
3 contract GeoToken is ERC20 {
4     constructor() ERC20("GeoToken", "GTK") {
5         // Ajuste a quantidade inicial de tokens se necessário
6         _mint(msg.sender, 100 ** decimals());
7     }
8 }
```

Código 2: Exemplo de código

- **Função ‘constructor’:** Inicializa o contrato, atribuindo 100 de *tokens* ao criador do contrato.
- **Importação do *OpenZeppelin*:** Utilizamos uma biblioteca padrão para garantir segurança e compatibilidade.

## 5 Tokenização

No contexto de *blockchains*, *tokens* são representações digitais de valor, ativos ou direitos que podem ser transferidos, negociados ou utilizados dentro de um sistema descentralizado. Eles são criados e gerenciados por contratos inteligentes e podem desempenhar diversas funções, como meio de pagamento, acesso a serviços ou registro de propriedade Antonopoulos & Wood (2018).

*Tokens* diferem das moedas digitais nativas, como o *Ether* no *Ethereum*, pois são implementados como abstrações sobre uma *blockchain* existente, com regras e funcionalidades definidas por contratos inteligentes.

## 5.1 Tipos de *tokens*

Os *tokens* são classificados em duas categorias principais:

- **Fungíveis:** São intercambiáveis, como dinheiro ou commodities (por exemplo, um token de 1 unidade é igual a outro de 1 unidade).  
Exemplos: *tokens* ERC-20, como o USDT ou DAI.
- **Não Fungíveis (NFTs):** Representam ativos únicos, como obras de arte digitais ou propriedades imobiliárias.  
Exemplos: *tokens* ERC-721, como *CryptoKitties* e *Bored Ape Yacht Club*.

## 5.2 Padrões de Implementação

Os padrões de *tokens* são fundamentais para garantir a interoperabilidade e segurança na *blockchain*. Alguns dos padrões mais conhecidos incluem:

1. **ERC-20:**
  - O padrão para *tokens* fungíveis no *Ethereum*.
  - Define funções como *transfer*, *approve* e *balanceOf*.
  - Facilita a criação de *tokens* que podem ser negociados em *exchanges* e integrados com carteiras.
2. **ERC-721**
  - O padrão para *tokens* não fungíveis (NFTs).
  - Inclui funções para verificar a propriedade e transferir ativos únicos.
3. **ERC-1155**
  - Um padrão híbrido para suportar tanto *tokens* fungíveis quanto não fungíveis.
  - Ideal para jogos e colecionáveis digitais que envolvem múltiplos tipos de ativos.
4. **ERC-1400**
  - Padrão de *token* de segurança, que inclui funções de controle e conformidade regulatória.

## 5.3 Funcionamento e Utilidade

Os *tokens* são gerenciados por contratos inteligentes, que definem suas regras de emissão, transferência e uso. Sua utilidade é ampla:

- **Representação de Ativos:** Propriedades físicas, como imóveis ou commodities, podem ser *tokenizadas*, permitindo a negociação fracionada e simplificada desses ativos Antonopoulos & Wood (2018)
- **Moedas Digitais:** Muitos *tokens* são usados como formas de moeda digital dentro de ecossistemas descentralizados.
- **Governança:** *tokens* podem representar direitos de voto em organizações descentralizadas (DAOs).
- **Utilidade e Acesso:** Alguns *tokens* oferecem acesso a serviços, como armazenamento descentralizado, ou funcionam como meios de pagamento dentro de uma aplicação específica.
- **Finanças Descentralizadas (DeFi):** *tokens* desempenham um papel central em protocolos DeFi, como *pools* de liquidez, empréstimos e *staking*.

A tokenização representa um dos avanços mais significativos proporcionados pela tecnologia *blockchain*, permitindo a digitalização de ativos e direitos, além de simplificar transações. Com padrões bem estabelecidos, como ERC-20 e ERC-721, os *tokens* oferecem versatilidade e potencial para transformar diversos setores, desde finanças até artes e entretenimento.

## 6 Ferramentas de desenvolvimento

O desenvolvimento de contratos inteligentes e *tokens* em plataformas *blockchain* como *Ethereum* é facilitado por ferramentas especializadas. Neste projeto, foram utilizadas as ferramentas *Ganache* e *Truffle*, que desempenham papéis fundamentais no ciclo de desenvolvimento, teste e implantação de contratos inteligentes.

### 6.1 Ganache

*Ganache* é uma ferramenta desenvolvida pela *Truffle* Suite que cria uma *blockchain* local para o desenvolvimento e teste de contratos inteligentes.

#### 6.1.1 Principais Características

1. *blockchain* Local
  - Simula uma rede *Ethereum* em execução localmente.
  - Permite testar contratos inteligentes em um ambiente controlado e sem custos com "gas".
2. Facilidade de Teste
  - Fornece várias contas pré-configuradas com saldo de *Ether* fictício, facilitando a execução de transações.
  - Registra todas as transações para análise e depuração.
3. Interface Intuitiva
  - Disponível em versões de linha de comando (*Ganache CLI*) e interface gráfica (*Ganache UI*).
  - A interface gráfica exibe detalhes de transações, blocos e eventos em tempo real.

#### 6.1.2 Uso no projeto

*Ganache* foi usado para:

- Testar o contrato em uma *blockchain* simulada antes de sua implantação em uma rede pública como *Ethereum*.
- Monitorar o comportamento de transações e eventos emitidos pelos contratos inteligentes.

### 6.2 Truffle

*Truffle* é um framework robusto para desenvolvimento de contratos inteligentes no *Ethereum*, projetado para simplificar tarefas como compilação, teste e implantação.

#### 6.2.1 Principais funcionalidades

1. Gerenciamento de Projetos

- Estrutura os projetos com diretórios predefinidos para contratos (*contracts/*), migrações (*migrations/*) e testes (*test/*).
2. Compilação de Contratos
    - Automatiza a compilação de contratos escritos em **Solidity**, gerando bytecode e arquivos ABI (Interface Binária de Aplicação).
  3. Migração
    - Facilita a implantação de contratos na **blockchain** com scripts de migração.
  4. Execução de Testes
    - Permite a criação de testes automatizados usando frameworks como *Mocha* e *Chai* para garantir a funcionalidade dos contratos.
  5. Integração com **Ganache**
    - Conecta-se à **blockchain** local do **Ganache** para testar contratos em um ambiente simulado.

### 6.2.2 Comandos essenciais no *Truffle*

- **Truffle init**: Inicializa um novo projeto com a estrutura básica.
- **Truffle compile**: Compila os contratos escritos em **Solidity**.
- **Truffle migrate**: Implanta os contratos na rede configurada.
- **Truffle test**: Executa os testes automatizados.

### 6.2.3 Uso no projeto

*Truffle* foi usado para:

- Organizar o projeto, mantendo uma estrutura clara para os contratos e *scripts* de migração.
- Automatizar a implantação do **token** ERC-20.

## 7 Especificação e desenvolvimento do projeto

Passo a passo para o desenvolvimento do projeto:

1. Bibliotecas do **Npm** e **Node** (versão)

[H]

Figura 1: A sample figure

2. Instalando o *Truffle*

```
1 $ npm install -g truffle
```

Código 3: Instalando truffle

3. Instalando o *Ganache*

```
1 $ npm install -g ganache
```

Código 4: Instalando ganache

4. Criando a estrutura de diretórios e inicializando com o *Truffle*

```
1 $ mkdir GeoToken
2 $ cd GeoToken
3 $ truffle init
4 $ npm init
```

Código 5: Criando diretórios



5. Escolher o Padrão do *Token*

- Use ERC-20 para um *token* fungível.
- Use ERC-721 para um *token* não fungível (NFT).

Iremos utilizar no exemplo a ERC-20.

## 6. Escrever o contrato inteligente

Instale a biblioteca *OpenZeppelin*.

```
1 $ npm install @OpenZeppelin/contracts
```

Código 6: Instalando biblioteca

## 7. Criando arquivo do contrato

Dentro do diretório **contracts** crie um arquivo chamado **GeoToken.sol**.

```
1 $ touch contracts/GeoToken.sol
```

Código 7: Instalando biblioteca

8. Escreva o contrato usando *Solidity*

```
1 import "@OpenZeppelin/contracts/token/ERC20/ERC20.sol";
2
3 contract GeoToken is ERC20 {
4     constructor() ERC20("GeoToken", "GTK") {
5         // Ajuste a quantidade inicial de tokens se necessário
6         _mint(msg.sender, 100 ** decimals());
7     }
8 }
```

Código 8: Código do contrato Solidity

## 9. Configuração do compilador

Confira se o arquivo **truffle-config.ts** está com o trecho que configura o *Solidity* Compiler.

```
1     compilers: {
2       solc: {
3         version: "0.8.21",
4         settings: {
5           optimizer: {
6             enabled: true,
7             runs: 200
8           }
9         }
10      }
11 }
```

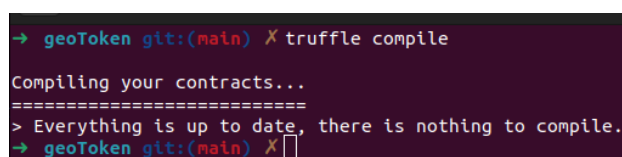
Código 9: Configuração do truffle

## 10. Compilar o contrato

Comando para compilar o contrato.

```
1 $ truffle compile
```

Código 10: Compilando contrato



```
→ geoToken git:(main) X truffle compile
Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.
→ geoToken git:(main) X
```

Figura 2: Compilando o contrato

### 11. Criando contrato de migração

Vamos criar o script de migração para implementar o *blockchain* local. Para isso, vamos criar o arquivo `migrations/2_deploy_contracts.js`

```
1 const GeoToken = artifacts.require("GeoToken");
2
3 module.exports = function (deployer) {
4     deployer.deploy(GeoToken);
5 };
```

Código 11: Código de migração

### 12. Iniciando o *Ganache*

Vamos passar por padrão o `gasLimit` de 12000000, a porta padrão 7574 e o id 1337.

```
1 $ ganache --gasLimit 12000000 --port 7545 --networkId 1337
```

Código 12: Iniciando o *Ganache*

### 13. Implementando o contrato

```
1 $ truffle migrate
```

Código 13: Implementando o contrato

```
→ geoToken git:(main) X truffle migrate
Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Starting migrations...
=====
> Network name: 'development'
> Network id: 1337
> Block gas limit: 12000000 (0xb71b00)

2_deploy_contracts.js
=====
Replacing 'GeoToken'
-----
> transaction hash: 0xd9e5214224bdd49ac6aff81a2db1e925f7fff31dd1caab971af3111bcd7ac50
> Blocks: 0
> contract address: 0x862e2AD3F1a78782C482921E068689Fc53b76120
> block number: 1
> block timestamp: 1733931358
> account: 0xDcf12a996832c77445d92f4D84Bf8119a37329C5
> balance: 999.98901806
> gas used: 549097 (0x860e9)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.01098194 ETH

> Saving artifacts
-----
> Total cost: 0.01098194 ETH

Summary
=====
> Total deployments: 1
> Final cost: 0.01098194 ETH

→ geoToken git:(main) X
```

Figura 3: truffle migrate

### 14. Testando o token com o *Truffle* console

```
1 $ truffle console
```

Código 14: Implementando o contrato

### 15. Interaja com o contrato recuperando o token

```
1 let instance = await GeoToken.deployed();
```

Código 15: Recuperando os tokens



detalhado sobre os contratos inteligentes e a *Ethereum* Virtual Machine (EVM), bem como as ferramentas utilizadas para o desenvolvimento e teste de soluções baseadas na tecnologia *blockchain*.

A implementação prática de um token ERC-20 demonstrou como padrões estabelecidos garantem segurança, eficiência e interoperabilidade na criação de ativos digitais. Ferramentas como *Ganache* e *Truffle* foram essenciais para simular o ambiente de *blockchain* e validar o comportamento do contrato antes de sua implantação em uma rede pública.

Os resultados obtidos mostram que o desenvolvimento de *tokens* no *Ethereum* oferece inúmeras possibilidades, desde a representação de ativos físicos e digitais até a criação de economias descentralizadas e sustentáveis. Contudo, desafios relacionados a custos de gas, escalabilidade e segurança ainda precisam ser continuamente enfrentados.

Como trabalhos futuros, sugere-se explorar a implementação de *tokens* não fungíveis (NFTs) usando o padrão ERC-721 e analisar seu impacto em setores como arte digital, jogos e gestão de identidades digitais.

## Referências

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