PROPOSTA DE ARQUITETURA PARA APLICAÇÕES DESCENTRALIZADAS *

Paulo Cesar Martins Citron† Jorge Luis Boeira Bavaresco‡

28 de julho de 2022

Resumo

A Web 3.0 e as aplicações descentralizadas possibilitam a criação de sistemas complexos sem a necessidade de servidores e autoridades gerenciadoras. O objetivo deste trabalho é contribuir para o desenvolvimento dessas aplicações, através da proposição de uma arquitetura para sua construção. Foi possível testar a solução, implementando a arquitetura proposta em uma aplicação para gerência e assinatura de contratos digitais, utilizada como estudo de caso. Os resultados da implementação confirmam que a modelagem da arquitetura é funcional, e sustentam a visão de que é possível construir sistemas visando sua autonomia e proteção da privacidade.

Palavras-chaves: Aplicações Descentralizadas. Blockchain. Ethereum. Web 3.0.

1 INTRODUÇÃO

Desde seu surgimento, a internet evolui a comunicação e interação humana, passando por diversas transformações, e constituindo novos modelos de negócio. Através da internet, o comércio, a indústria e a sociedade se reinventam constantemente, e para propiciar tais mudanças, novas tecnologias surgem para resolver problemas antes insolucionáveis.

Um dos maiores problemas do modelo atual é a coleta e centralização de dados, ocasionando em constante conflito entre experiência e privacidade. Contudo, segundo Wood (2018), o qual propõe um conceito denominado Web 3.0, tecnologias como *Blockchain* e redes *peer-to-peer*[1] possibilitaram fazer tudo que é feito hoje sem servidores e nem

*Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, na cidade de Passo Fundo, em 2022.

†paulo.citron@proton.me

‡jorgebavaresco@ifsul.edu.br

[1] Arquitetura de redes de computadores onde a comunicação é feita diretamente pelos nós da rede

autoridades para gerenciar o fluxo de informações. Por meio de aplicações descentralizadas, é possível construir redes de informações que não ultrapassam os limites de privacidade dos usuários, além de eliminar a necessidade de autoridades certificadoras.

Este trabalho tem por objetivo definir uma proposta arquitetural para construção de aplicações descentralizadas, visando contribuir em trabalhos que buscam aplicar esse conceito na resolução de problemas relacionados. Através de um estudo de caso, utilizando a arquitetura proposta, serão avaliados os resultados obtidos através de sua aplicação.

O estudo de caso consiste em uma aplicação de Contratos Inteligentes, utilizando a plataforma Ethereum, para construção de um sistema de assinatura digital de acordos. Os contratos serão implementados em Solidity, e serão acessados através de uma aplicação web construída utilizando React, o qual utilizará a biblioteca Ethers.js para comunicação com a rede Blockchain. As transações serão realizadas utilizando a extensão Metamask no browser Firefox. Tanto a aplicação web quanto a rede Ethereum serão executados em um Container Linux, utilizando Docker.

O artigo está organizado nas seguintes seções: A seção 2 apresenta a fundamentação teórica utilizada no desenvolvimento do trabalho, apresentando os conceitos e tecnologias. A seção 3 detalha a arquitetura proposta, bem como o estudo de caso e metodologia proposta. A seção 4 apresenta o desenvolvimento do estudo de caso, além dos resultados obtidos. Ao final, na seção 5, estão as considerações finais, apresentando possibilidades de trabalhos futuros e conclusões emergentes dos resultados.

2 REFERENCIAL TEÓRICO

Nesta seção será apresentada a fundamentação teórica que sustenta a modelagem e aplicação das tecnologias relacionadas.

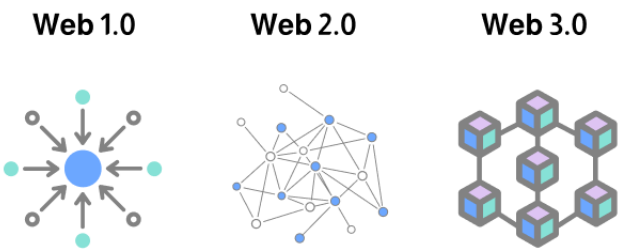
2.1 Web 3.0 e Aplicações Descentralizadas

A Web 3.0, inicialmente proposta por Wood (2018), é um conjunto inclusivo de protocolos para fornecer meios de construção de aplicativos, substituindo as tecnologias tradicionais da Web, e possibilitando a descentralização.

O conceito foi originado através da constante evolução de tecnologias como Blockchain, possibilitando criar aplicações descentralizadas (DApp), as quais, após implantadas, não precisam de manutenção ou governança de seus criadores. Portanto, essas aplicações permaneceriam operacionais sem qualquer intervenção humana, formando uma Organização Autônoma Descentralizada (DAO) (GREEN, 2016).

A Figura 1 apresenta a evolução da internet, segundo Nath, Dhar e Basishtha (2014) consistindo em 3 fases, partindo de uma rede estática de leitura, centralizada e de baixa portabilidade (Web 1.0), passando por uma rede dinâmica de leitura e escrita, distribuída em serviços e de média portabilidade (Web 2.0) e chegando em uma rede inteligente, interativa e descentralizada (Web 3.0).

Figura 1 – Evolução da Web



Fonte: adaptado de Neto (2022)

Utilizando dos protocolos da Web 3.0, este trabalho busca propor uma arquitetura para construção de aplicações descentralizadas.

2.2 Tecnologias utilizadas

Nesta seção serão apresentadas as fundamentações teóricas que embasam a implementação das tecnologias utilizadas.

2.2.1 Contratos Inteligentes

Contratos Inteligentes (ou *Smart Contracts*, do original) são contratos incorporados ao hardware e software, de forma a tornar a quebra desses contratos inviável ao infrator Szabo (1996). Com o uso de contratos inteligentes, pode-se substituir a necessidade de participação de uma terceira parte em uma transação ou acordo, utilizando a tecnologia como meio.

Segundo Crosby et al. (2016), em tradução livre: "O serviço de certificação de documentos auxilia na Prova de Titularidade (quem é o autor), Prova de Existência (em determinado momento) e Prova de Integridade (não adulterada) dos documentos". Utilizando Contratos Inteligentes, a certificação seria gerida de forma automática, sem necessidade de uma autoridade. Neste trabalho, é explorada a aplicação deste conceito na construção de contratos inteligentes para armazenamento e processamento de dados de aplicações descentralizadas.

2.2.2 Blockchain

O Blockchain, segundo Crosby et al. (2016), consiste-se essencialmente em um banco de dados distribuído de registros (ou razão pública) de todos os eventos digitais que foram executados e compartilhados entre as partes participantes, sendo todos os eventos verificados pelo consenso da maioria dos participantes do sistema.

Esses eventos denominam-se blocos (em formato de hash), os quais estão ligados uns aos outros (como uma cadeia) em ordem cronológica linear, com cada bloco contendo o hash do bloco anterior (CROSBY et al., 2016).

Segundo Norton (2017), a Blockchain torna possível garantir a confiabilidade no armazenamento de dados. A ligação dos blocos os torna imutáveis, visto que se um bloco fosse alterado (gerando uma nova hash) a cadeia seria quebrada. Este trabalho utiliza a tecnologia Blockchain para o armazenamento dos dados e contratos inteligentes da aplicação descentralizada.

2.2.3 Ethereum

Ethereum é uma plataforma de desenvolvimento de aplicações descentralizadas, possibilitando sua implementação em redes *peer-to-peer* públicas e privadas. Utilizando algoritmos de *Proof of Work*, a plataforma permite a construção de contratos inteligentes, os quais utilizam o poder de processamento da rede para execução de funções contidas nesses contratos. Os responsáveis pelo processamento são recompensados através da criptomoeda da plataforma (ETH), advinda da cobrança de uma taxa na realização da transação (evento) na rede, denominada Gas (ETHEREUM, 2022).

Os contratos são escritos em Solidity, uma linguagem de alto nível, orientada a objetos e de tipagem estática desenvolvida para a criação dos mesmos na *Ethereum Virtual Machine* (EVM) (SOLIDITY, 2022). Este trabalho utiliza a plataforma Ethereum, tendo a lógica desenvolvida em Solidity para gerência dos contratos digitais.

2.2.4 Truffle Suite

Truffle Suite consiste em uma série de ferramentas para desenvolvimento de aplicações descentralizadas, utilizando redes de Blockchain baseadas na EVM. Uma das ferramentas da Truffle Suite é o Ganache, uma Blockchain pessoal que possibilita o desenvolvimento e teste de aplicações descentralizadas em um ambiente determinístico controlado (TRUFFLESUITE, 2022).

Outra ferramenta é o Truffle, um ambiente de desenvolvimento que utiliza a EVM, além de um framework de testes e pipeline de compilação de contratos inteligentes para artefatos. Neste trabalho, o Ganache é utilizado para execução conteinerizada de uma Blockchain, possibilitando a implantação dos contratos inteligentes testados e compilados utilizando Truffle (TRUFFLESUITE, 2022).

2.2.5 Metamask

Metamask é uma carteira multi-criptomoeda e gateway de conexão para aplicações descentralizadas que utilizam a plataforma Ethereum (METAMASK, 2022).

Disponibilizada como extensão para navegadores web e aplicativo mobile, a Metamask permite a gerência de tokens e endereços, além de possibilitar transacionar em uma rede Blockchain. Este trabalho utiliza a Metamask para realização das transações através da aplicação web.

2.2.6 React

React é um framework javascript criado por engenheiros do Facebook para resolver desafios envolvendo o desenvolvimento de interfaces de usuário complexas com conjuntos de dados que mudam frequentemente (GACKENHEIMER, 2015).

No React, existe uma estrutura chamada componente, a qual oferece uma separação de itens de uma página web que possuem comportamentos específicos, ou que precisam

ser recarregados em diferentes momentos (possibilitando carregar apenas um componente específico). Utilizando componentes, a criação e manutenção de uma aplicação web dinâmica é facilitada, utilizando de suas estruturas de reaproveitamento (REACT, 2022).

Além disso, o React possui um conceito chamado Virtual DOM, o qual, através de uma representação virtual do DOM real da aplicação, possibilita abstrair a manipulação de atributos, manipulação de eventos e atualização manual do DOM real da aplicação, apenas gerindo estados de seus elementos.

Este trabalho utiliza o React para construção da interface do usuário da aplicação web, acessada pelos usuários finais para interação e visualização dos contratos digitais. Além disso, o trabalho utilizará o Vite, uma ferramenta de construção que visa fornecer uma experiência de desenvolvimento mais rápida e enxuta para projetos web modernos (VITE, 2022).

2.2.7 Ethers

Ethers é uma biblioteca Javascript completa e compacta para interagir com o Ethereum e seu ecossistema. Através dele, é possível acessar métodos e informações dos protocolos Web 3.0, através de uma aplicação web, utilizando a linguagem Javascript (ETHERS, 2022).

Utilizando Ethers juntamente ao React, a interação do usuário com as funções dos contratos é possibilitada, gerando transações controladas através da carteira (Metamask). Através de Artefatos, contratos inteligentes escritos em Solidity e compilados em um formato JSON, o Ethers fornece objetos e funções para execução e controle das informações. Este trabalho utiliza o Ethers para construção das funcionalidades da aplicação web que possibilitam a interação com a rede Ethereum (ETHERS, 2022).

2.2.8 Docker

Segundo Anderson (2015) Docker é uma tecnologia de virtualização de contêineres, como uma máquina virtual muito leve. Seu uso possibilita que os softwares funcionem corretamente, independente do sistema operacional utilizado.

Através de sua utilização, além de tornar o sistema desenvolvido portátil (pode facilmente ser migrado de plataforma), torna mais fácil a gestão de recursos de sua hospedagem. Este artigo utiliza o Docker, através de contêineres Linux, como sistema virtual para execução da aplicação web, bem como da rede privada com Ganache, facilitando seu desenvolvimento e configuração (DOCKER, 2022).

3 ARQUITETURA

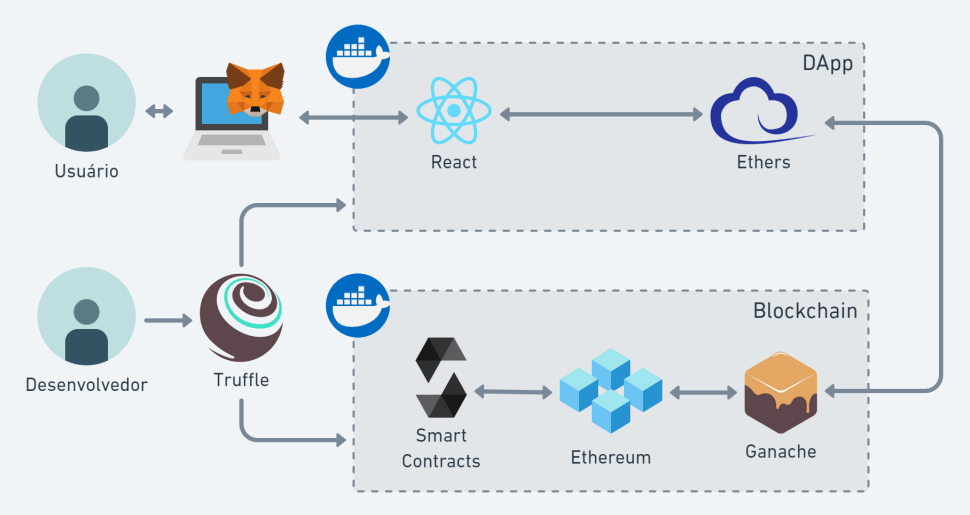
Nesta seção a arquitetura proposta é apresentada, bem como sua modelagem e o funcionamento do estudo de caso.

3.1 Modelagem da arquitetura

Utilizando das tecnologias e fundamentação teórica, a Figura 2 ilustra a arquitetura proposta para construção de uma aplicação descentralizada. Através de dois contêineres Linux, executados utilizando Docker, a Blockchain Ethereum local (Ganache) é executada, bem como a DApp (aplicação web). O desenvolvedor cria e executa os testes utilizando

o Truffle, o qual possibilita realizar o deploy[2] dos Contratos Inteligentes criados para o Ganache.

Figura 2 – Arquitetura proposta



Fonte: do autor, 2022

O usuário, através de um navegador web com a extensão Metamask instalada, acessa o DApp, conectando sua conta na rede da Blockchain. A rede possui o(s) contrato(s) escritos em Solidity, com suas funções acessíveis através da comunicação entre a biblioteca Ethers, munida dos artefatos compilados utilizando Truffle, e o Ganache, tendo as transações aprovadas através da Metamask.

O DApp, através de componentes criados utilizando React, exibirá os dados através de consultas aos métodos públicos dos contratos, e enviará dados através de formulários que, utilizando Ethers, interagem com os Contratos Inteligentes através da conta conectada.

3.2 Estudo de caso

Para aplicação da arquitetura, foi idealizado um estudo de caso para criação, gerência e assinatura de contratos digitais. O contrato pode ser criado por uma conta, a qual automaticamente o assina, e define mais quantos assinantes serão necessários. Além disso, cada conta pode assinar somente uma vez, garantindo a integridade das assinaturas.

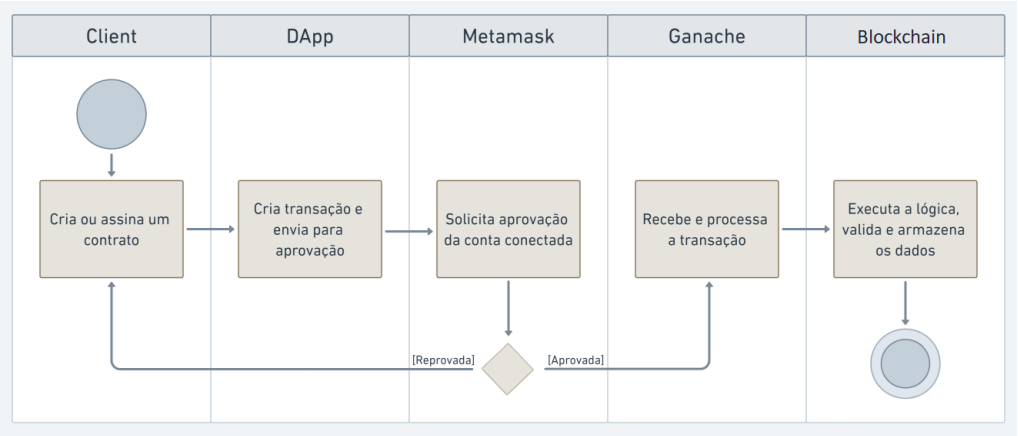
Para o estudo de caso, foi idealizado um fluxo de assinatura pública, ou seja, qualquer conta pode assinar qualquer contrato criado, contanto que o contrato ainda precise de assinaturas, e a conta ainda não o tenha assinado.

A Figura 3 demonstra o funcionamento da arquitetura implementada. Quando o usuário realizar uma interação (tanto de criação quanto de assinatura), o DApp criará uma transação, a qual será enviada a Metamask para aprovação. Caso aprovada, a transação é enviada ao Ganache, o qual irá receber e processar para a rede Blockchain, que executará

[2] implantação do software em um ambiente

o Contrato Inteligente com sua lógica e validações, por fim armazenando os dados da transação em um Bloco processado pela rede.

Figura 3 – Diagrama de atividade do estudo de caso



Fonte: do autor, 2022

O contrato consiste em um conteúdo em formato string, data de criação, identificação do criador, quantidade de assinaturas requeridas, e quantidade de assinaturas realizadas. As assinaturas serão armazenadas usando uma entidade chamada Signatário, tendo o contrato uma ligação para um array de signatários. As assinaturas conterão uma string identificadora, data da realização e endereço da carteira do assinante. Serão implementados, também, eventos de criação e assinatura dos contratos, possibilitando visualização na Blockchain.

A aplicação web do estudo de caso consistirá em SPA (*Single Page Application*), com um componente para adição de contratos, um para listagem dos contratos, e um componente para visualização dos dados do contrato, contendo seus dados básicos, os signatários e um botão para realizar assinatura, caso ainda não o tenha assinado. Além disso, aplicação web contém componentes com informações da conta conectada, como endereço e quantidade de criptomoedas.

4 DESENVOLVIMENTO

Nesta seção é demonstrada, através de trechos de código[3], a aplicação da arquitetura proposta no estudo de caso, bem como sua construção. Além disso, o funcionamento e os resultados de sua implementação também são apresentados.

4.1 Construção do Contrato Inteligente

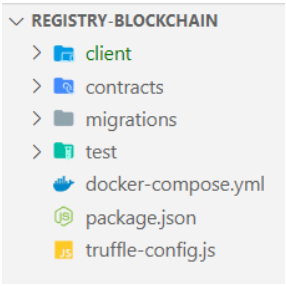
Na construção do estudo de caso, foi utilizada uma técnica de desenvolvimento denominada *Test Driven Development* (TDD), que segundo Anwer et al. (2017), é uma técnica ágil que consiste em construir o software em pequenas iterações contínuas, escrevendo

[3] Código completo disponível em: <https://github.com/paulowsky/registry-blockchain>

testes automatizados primeiro, implementando o código para que passe nesses testes, e refatorando o código a medida que cresce para que suporte complexidades superiores.

A estrutura do projeto, conforme ilustra a Figura 4, se dá por uma pasta denominada "client", a qual contém a aplicação web, uma pasta chamada "contracts", que contém os contratos inteligentes escritos em Solidity, uma pasta denominada "migrations", contendo a estrutura para possibilitar o deploy dos contratos, e a pasta "test", que por sua vez contém os testes automatizados. Além disso, existe um arquivo com a implementação dos containers Docker, um arquivo de dependências Node e um arquivo de configuração do Truffle.

Figura 4 – Estrutura do projeto



Fonte: do autor, 2022

A configuração para execução do Ganache e da aplicação web em *container*, ilustrada na Figura 5(a), consiste em dois serviços, rodando em contêineres separados, ligados por uma rede em modo *bridge* para comunicação interna, e acessíveis externamente por uma porta mapeada.

No arquivo de configuração do Truffle, são definidos o diretório onde serão salvos os artefatos compilados, as configurações da dashboard, definições de rede e otimizações de execução, como demonstrado na Figura 5(b).

Figura 5 – Configuração do Docker Compose e do Truffle

```yaml
version: '2'
services:
  ganache:
    image: trufflesuite/ganache:v7.2.0
    hostname: ganache
    container_name: ganache
    ports:
      - 8545:8545
    command: --wallet.totalAccounts 10 --wallet.defaultBalance 100 --server.port 8545
    networks:
      - registry-network
  client:
    build: client/.
    hostname: client
    container_name: client
    ports:
      - '3000:3000'
    networks:
      - registry-network

networks:
  registry-network:
    driver: bridge
```

(a) Configuração do Docker Compose

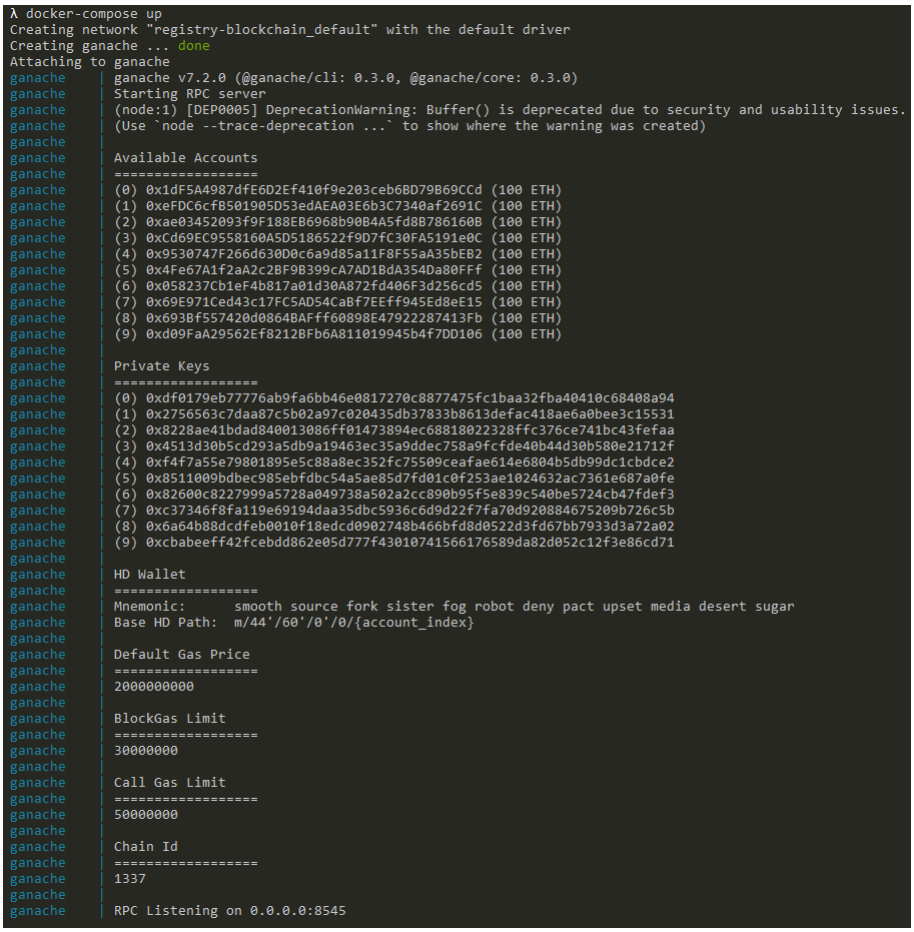
```javascript
module.exports = {
  contracts_build_directory: "./client/src/artifacts/",
  dashboard: {
    port: 24012
  },
  networks: {
    development: {
      host: process.env.HOST || "127.0.0.1",
      port: process.env.PORT || 8545,
      network_id: process.env.NETWORK_ID || "*"
    }
  },
  solc: {
    optimizer: {
      enabled: true,
      runs: 200
    }
  }
}
```

(b) Configuração do Truffle

Fonte: do autor, 2022

Ao executar o Docker Compose para iniciar os contêineres, o Ganache disponibiliza, nos logs de execução, as contas e suas chaves privadas, além dos valores padrão de Gas e porta em que está aberto para conexão RPC, conforme demonstrado na Figura 6.

Figura 6 – Execução do container do Ganache



Fonte: do autor, 2022

Seguindo o processo do TDD, os testes foram escritos antes de construir o sistema. Inicialmente, foi escrito um teste para validação do deploy, ilustrado na Figura 7, o qual possui um hook[4] que executa antes de cada teste, inicializando o artefato compilado e as contas que serão utilizadas nos outros testes. Além disso, o teste de validação do deploy foi implementado, validando o endereço do contrato na Blockchain.

[4] Trecho de código que executa antes ou após determinado evento acontecer

Figura 7 – Testes do Deploy

```
const Registry = artifacts.require('./Registry.sol')

contract('Registry', (accounts) => {
  before(async () => {
    this.registry = await Registry.deployed()

    const accounts = await web3.eth.getAccounts()
    const creator = accounts[0]
    const signer = accounts[1]
  })

  it('should deploy successfully', async () => {
    const address = await this.registry.address

    assert.notEqual(address, 0x0)
    assert.notEqual(address, '')
    assert.notEqual(address, null)
    assert.notEqual(address, undefined)
  })
```

Fonte: do autor, 2022

Para o funcionamento da aplicação, foram implementados dois testes, um para criação de um contrato, e um para assinatura do mesmo. A Figura 8(a) demonstra o teste de criação, que usa uma função que recebe o nome do criador do contrato, o conteúdo e a quantidade de assinaturas. Após isso, busca o contrato e seus assinantes, para efetuar as validações. A Figura 8(b) ilustra o teste de assinatura, que utiliza uma função que recebe o id do contrato e o nome do signatário, buscando e validando as informações. Ambos possuem seus eventos relacionados, também validados nos testes.

Figura 8 – Testes das funções dos Contratos Inteligentes

```
it('should be possible to create contract', async () => {
  const result = await this.registry.createContract(
    'creator',
    'test contract 2',
    3,
    { from : accounts[0] }
  )

  const contractsCount = await this.registry.contractsCount()

  const contractId = contractsCount - 1;
  const creatorSignatoryId = 0;

  const contract = await this.registry.contracts(contractId)

  const signatories = []
  for (let i=0; i < (contract.amountSigned.toNumber()); i++) {
    signatories.push(
      await this.registry.signatories(contractId, i)
    )
  }

  assert.equal(contractsCount.toNumber(), 1)

  assert.equal(contract.content, 'test contract 2')

  assert.equal(signatories[creatorSignatoryId].name, 'creator')
  assert.equal(
    signatories[creatorSignatoryId].signatoryAddress,
    accounts[0]
  )

  const event = result.logs[0].args
  assert.equal(event.contractId.toNumber(), contractId)
})
```

(a) Teste de criação de contrato

```
it('should be possible to sign contract', async () => {
  const result = await this.registry.signContract(
    0,
    'signer',
    { from : accounts[1] }
  )

  const contractId = 0;
  const creatorSignatoryId = 0;
  const signatoryId = 1;

  const contract = await this.registry.contracts(contractId)

  const signatories = []
  for (let i=0; i < contract.amountSigned.toNumber(); i++) {
    signatories.push(
      await this.registry.signatories(contractId, i)
    )
  }

  assert.equal(signatories[creatorSignatoryId].name, 'creator')
  assert.equal(
    signatories[creatorSignatoryId].signatoryAddress,
    accounts[0]
  )
  assert.equal(signatories[signatoryId].name, 'signer')
  assert.equal(
    signatories[signatoryId].signatoryAddress,
    accounts[1]
  )

  const event = result.logs[0].args
  assert.equal(event.contractId.toNumber(), contractId)
})
```

(b) Teste de assinatura de contrato

Fonte: do autor, 2022

A Figura 9 ilustra as entidades do contrato, bem como seus eventos. O contador "contractsCount" serve para geração de identificadores únicos para os contratos criados, que são representados pela entidade "Contract", e acessíveis por um *array* público. Existem também uma entidade "Signatory", representando os signatários, que são salvos em um *array* público em que o identificador é o mesmo do identificador do contrato, permitindo associá-los com facilidade. Além disso, existem dois eventos, um para criação e outro para assinatura, ambos salvando o contrato em questão, o endereço da conta que efetuou a transação e a data.

Figura 9 – Entidades, estruturas e eventos do Contrato Inteligente

```
pragma solidity >=0.5.16;
pragma experimental ABIEncoderV2;

contract Registry {
  uint public contractsCount = 0;

  struct Signatory {
    address signatoryAddress;
    string name;
    uint datetimeSigned;
  }

  struct Contract {
    string content;
    uint datetimeCreated;
    uint creatorSignatoryId;
    uint amountRequiredSignatures;
    uint amountSigned;
  }
```

(a) Entidades

```
  mapping(uint => Contract) public contracts;
  mapping(uint => Signatory[]) public signatories;

  event ContractCreated(
    uint contractId,
    uint datetimeCreated,
    address creator
  );

  event ContractSigned(
    uint contractId,
    uint datetimeSigned,
    address signer
  );

  constructor() public {}
}
```

(b) Estruturas públicas e eventos

Fonte: do autor, 2022

No Contrato Inteligente, foi codificada a função para criação de contratos, conforme ilustrado na Figura 10, a qual inicializa o contrato no *array* público, usando a tipagem "storage", além de também criar um objeto do tipo signatário ao criador do contrato, que também o assinará. Os mesmos são armazenados na Blockchain, e o evento de criação é emitido, por fim incrementado o contador de contratos.

Figura 10 – Função de criação de um contrato

```
function createContract(
  string memory _name,
  string memory _content,
  uint _amountSignatories
) public {
  uint _datetimeCreated = block.timestamp;
  uint _creatorSignatoryId = 0;

  Contract storage newContract = contracts[contractsCount];
  newContract.content = _content;
  newContract.datetimeCreated = _datetimeCreated;
  newContract.creatorSignatoryId = _creatorSignatoryId;
  newContract.amountRequiredSignatures = _amountSignatories + 1;
  newContract.amountSigned = 1;

  Signatory memory _signatory = Signatory({
    signatoryAddress: msg.sender,
    name: _name,
    datetimeSigned: _datetimeCreated
  });

  signatories[contractsCount].push(_signatory);

  emit ContractCreated(contractsCount, _datetimeCreated, msg.sender);

  contractsCount++;
}
```

Fonte: do autor, 2022

Para a assinatura do contrato, também foi implementada uma função, conforme

demonstrado na Figura 11, a qual verifica três condições antes de realizar a assinatura:

1. Se o identificador de contrato recebido é válido;
2. Se o contrato identificado ainda precisa de assinaturas;
3. Se a conta que está realizando a transação ainda não assinou o contrato.

Caso qualquer dessas verificações falhe, é emitido um erro e a transação não é concluída. Após as verificações, a conta é armazenada como signatário, sendo adicionada ao array relacionado ao contrato, ao fim emitindo o evento de assinatura.

Figura 11 – Função de assinatura de um contrato

```
function signContract(uint _contractId, string memory _name) public {
  // check if id is valid
  require(_contractId >= 0 && _contractId < contractsCount);

  Contract storage _contract = contracts[_contractId];

  // check if the contract need more signatures
  require(_contract.amountRequiredSignatures > _contract.amountSigned);

  bool _senderNotSigned = true;
  for (uint i=0; i < _contract.amountSigned; i++) {
    if (signatories[_contractId][i].signatoryAddress == msg.sender) {
      _senderNotSigned = false;
      break;
    }
  }
  require(_senderNotSigned);

  uint _datetimeSigned = block.timestamp;

  Signatory memory _signatory = Signatory({
    signatoryAddress: msg.sender,
    name: _name,
    datetimeSigned: _datetimeSigned
  });

  signatories[_contractId].push(_signatory);

  _contract.amountSigned++;

  emit ContractSigned(_contractId, _datetimeSigned, msg.sender);
}
```

Fonte: do autor, 2022

Para a implantação do contrato, foi criado um arquivo de migração para possibilitar salvá-lo na Blockchain, conforme ilustra a Figura 12. O deploy ocorre em uma transação, a qual é feita através do Truffle, que possui suporte ao formato de migrações, que permite versionar as alterações do artefato salvo na Blockchain.

Figura 12 – Configuração de migração

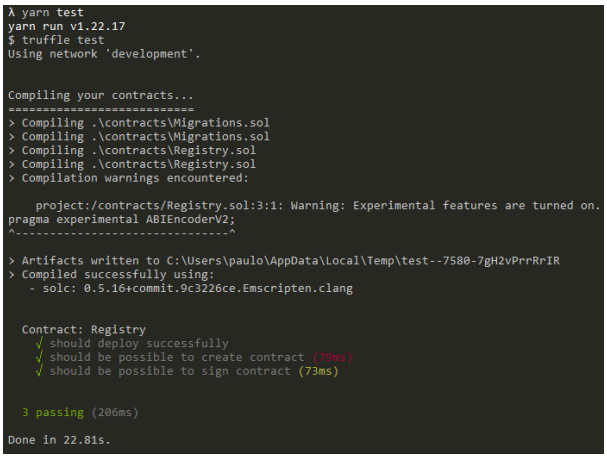
```
var Registry = artifacts.require("./Registry.sol")

module.exports = function(deployer) {
  deployer.deploy(Registry)
}
```

Fonte: do autor, 2022

Ao executar os testes, como demonstrado na Figura 13, o resultado é positivo, indicando que o funcionamento do código implementado está correto. O uso do TDD impõe obrigatoriamente uma cobertura de testes saudável, uma vez que para implementação de cada parte lógica do software, um teste automatizado deve ser codificado com antecedência. Isso possibilita adaptar e executar os testes a cada alteração da aplicação, mantendo sua consistência e assegurando seu funcionamento.

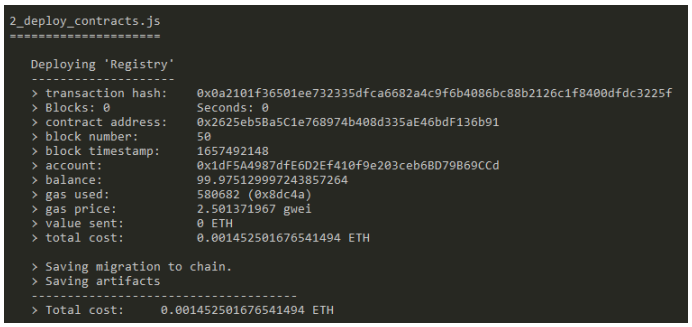
Figura 13 – Resultado da execução dos testes automatizados



Fonte: do autor, 2022

Para finalização da implementação da lógica do sistema, foi realizado o deploy dos contratos na Blockchain, conforme ilustrado na Figura 14, exibindo o resultado das transações, e o endereço do contrato implantado, o qual será utilizado posteriormente na aplicação web. Além disso, o custo (Gas) das transações é informado, bem como todas as informações do processamento.

Figura 14 – Resultado da execução das migrações



Fonte: do autor, 2022

4.2 Construção da aplicação web

Para construção da plataforma web, utilizando React e Ethers, foi utilizada uma abordagem de contexto, onde foram criados componentes que compartilham os dados e métodos centralizados.

Conforme demonstrado na Figura 15, foram utilizadas funções provenientes do React para criação de um contexto, o qual possui um *provider* com estados que armazenam quanto a conta conectada possui em Ether, o endereço da mesma e um objeto proveniente do Ethers, respectivamente. Além disso, é definido o endereço do contrato armazenado na Blockchain através da migration executada.

Figura 15 – Contexto de interação com a Blockchain

```
import React, { createContext, useState } from 'react'

import { ethers } from 'ethers'

import Registry from '@/artifacts/Registry.json'

export const Web3Context = createContext({})

export const Web3Provider = ({ children }) => {
  const contractAddress = '0x2625eb5ba5c1e768974b408d335ae46bdf136b91'

  const [balance, setBalance] = useState()
  const [address, setAddress] = useState()
  const [registry, setRegistry] = useState()
```

Fonte: do autor, 2022

Com o contexto configurado, foi criada uma função para gerenciamento e atualização da conta conectada através da Metamask, conforme ilustrado na Figura 16. A função "updateAccount" requisita as contas conectadas ao navegador, configuradas através da extensão, armazenando-as nos estados do contexto. Por fim, é criada uma instância do Contrato Inteligente, utilizando o Ethers, também salva no estado pertinente.

Figura 16 – Função para atualização da conta conectada na Metamask

```
const updateAccount = async () => {
  try {
    const [account] = await window.ethereum.request({
      method: 'eth_requestAccounts'
    })

    const provider = new ethers.providers.Web3Provider(window.ethereum)

    setBalance(ethers.utils.formatEther(await provider.getBalance(account)))
    setAddress(await provider._getAddress(account))

    setRegistry(
      new ethers.Contract(contractAddress, Registry.abi, provider.getSigner())
    )
  } catch (err) {
    console.error(err)
  }
}
```

Fonte: do autor, 2022

Para listagem dos contratos digitais, foi criada uma função para requisição das

indexações públicas na Blockchain, conforme demonstrado na Figura 17. Na implementação existem dois laços, onde percorrem os contratos e seus signatários, por fim retornando-os em um objeto.

Figura 17 – Função para listagem dos contratos digitais

```
const getContracts = async () => {
  try {
    let _contracts = []
    let _signatories = []

    const contractsCount = await registry.contractsCount()

    for (let i = 0; i < contractsCount; i++) {
      let currentContract = await registry.contracts(i)
      let _contractSignatories = []

      for (let j = 0; j < currentContract.amountSigned.toNumber(); j++) {
        let currentSignatory = await registry.signatories(i, j)
        _contractSignatories.push(currentSignatory)
      }
      _contracts.push(currentContract)
      _signatories.push(_contractSignatories)
    }

    return {
      contracts: _contracts,
      signatories: _signatories
    }
  } catch (err) {
    console.error(err)
  }
}
```

Fonte: do autor, 2022

Além disso, para interação com os contratos, foram criadas duas funções no contexto. A Figura 18(a) demonstra a implementação da função para criação de um contrato digital, a qual requisita o nome do assinante, salvando-o juntamente aos dados recebidos por parâmetro na Blockchain, e informando o usuário que a criação foi realizada com sucesso.

A Figura 18(b) ilustra a função de assinatura do contrato digital, onde o usuário informa o nome, e através do identificador do contrato recebido por parâmetro, a assinatura é realizada, por fim informando o usuário que a assinatura foi concluída com êxito.

Figura 18 – Funções de manipulação dos contratos digitais

```
const createContract = async (content, amountSignatories) => {
  try {
    const name = window.prompt('Type your name:')

    await registry.createContract(name, content, amountSignatories)

    window.alert('Contract created!')

    getContracts()
  } catch (err) {
    console.error(err)
  }
}
```

(a) Função para criação de um contrato

```
const signContract = async contractId => {
  try {
    const name = window.prompt('Type your name:')

    await registry.signContract(contractId, name)

    window.alert('contract signed!')

    getContracts()
  } catch (err) {
    console.error(err)
  }
}
```

(b) Função para assinatura de um contrato

Fonte: do autor, 2022

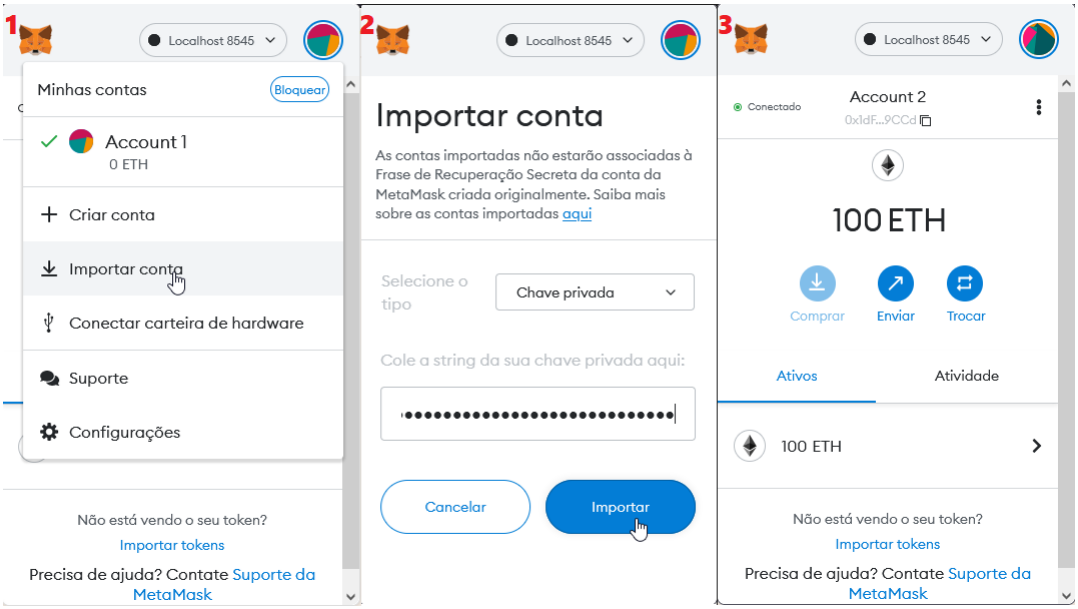
No contexto, foram implementadas funções para interação com a Blockchain

através do ethers, exportadas pelo *provider* para possibilitar o uso nos componentes. Além disso, um *Hook* foi criado para que seja possível importar as funções e estados do contexto, possibilitando seu uso nos componentes de maneira fácil através de Atribuição via Desestruturação (*Destructuring Assignment*) do Javascript.

4.3 Resultados

Ao acessar a plataforma web, é necessário primeiramente configurar as contas que serão utilizadas nas transações. Para tanto, foi utilizada a extensão Metamask, importando as contas de teste do ambiente de desenvolvimento através de sua chave privada que foi disponibilizada ao executar o Ganache, conforme ilustrado na Figura 19, onde a importação é feita através de três passos. Após a importação, a conta deve ser conectada via RPC (Remote Procedure Call) na rede de teste local, na porta **8545**, onde o Ganache está aguardando conexões. Por fim, a quantidade de Ether que a conta possui será exibida na Metamask, demonstrando que a importação foi realizada com sucesso.

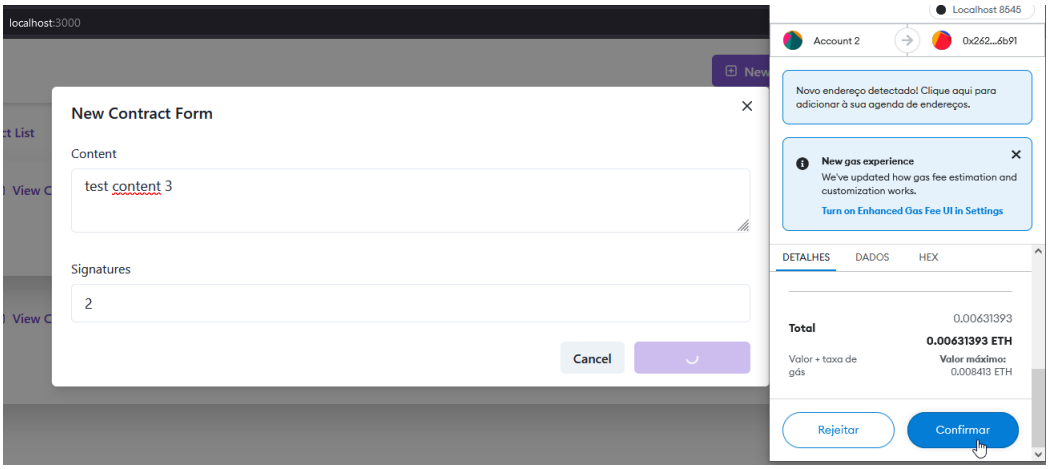
Figura 19 – Importação de conta de teste na Metamask



Fonte: do autor, 2022

Tendo a conta conectada, é possível acessar a aplicação, onde existe a interface de usuário para interação com as funções do Contrato Inteligente. A Figura 20 demonstra a criação de um contrato, onde foram informados o conteúdo e a quantidade de assinaturas requeridas, tendo a transação enviada para a Metamask para aprovação, com informações sobre o valor da taxa (Gas).

Figura 20 – Criação de um contrato através da aplicação web



Fonte: do autor, 2022

Com a transação confirmada, a Blockchain processa a mesma e salva na base, conforme demonstrado na Figura 21, onde é possível observar os *logs* provenientes do Ganache, com a hash gerada, o custo efetivo, o número do bloco em que a transação foi salva e a data de mineração.

Figura 21 – Log de transação do Ganache

```
ganache  | eth_sendRawTransaction
ganache  |
ganache  |   Transaction: 0xda16fdfd274895881fb315adabfae705513dc290ca39cb308eb11eaebbef0719
ganache  |   Gas usage: 211049
ganache  |   Block number: 54
ganache  |   Block time: Tue Jul 12 2022 16:30:06 GMT+0000 (Coordinated Universal Time)
ganache  |
ganache  | eth_call
ganache  | eth_call
ganache  | eth_call
ganache  | eth_call
ganache  | eth_call
ganache  | eth_blockNumber
ganache  | eth_getTransactionReceipt
ganache  | eth_getBlockByHash
```

Fonte: do autor, 2022

5 CONSIDERAÇÕES FINAIS

No presente artigo foi apresentada uma proposta arquitetural para construção de aplicações descentralizadas. Sua aplicação prática pôde ser observada na construção de um estudo de caso, trazendo resultados positivos no desenvolvimento.

Através de sua implementação, o estudo de caso demonstra que o uso do TDD como técnica de desenvolvimento oferece a possibilidade de realizar testes na parte lógica do sistema, como também uma documentação clara de como será seu funcionamento. Além disso, a execução automatizada destes testes acelera as validações e o fluxo de escrita de código.

Analisando os resultados, pode-se confirmar que é possível implementar uma aplicação descentralizada que abstém a necessidade de autoridades certificadoras, além de proteger os dados armazenados na Blockchain. O uso de ferramentas, como a Metamask, possibilita a portabilidade no uso destas tecnologias, auxiliando a construção de sistemas de alta usabilidade. Além disso, ferramentas de desenvolvimento como Docker, Truffle e Ganache facilitam o desenvolvimento das soluções.

A Web 3.0 vem demonstrando a viabilidade da proteção da privacidade dos usuários na internet, possibilitando a construção de aplicações inteligentes e descentralizadas. Os resultados da aplicação desta arquitetura sustentam essa visão, demonstrando o funcionamento prático deste conceito.

Portanto, para o desenvolvimento de melhorias futuras, propõem-se a implementação do suporte a arquivos na criação e manutenção de acordos, sendo possibilitados pela tecnologia IPFS (InterPlanetary File System). Não obstante, pode-se implementar um fluxo de assinatura mais complexo, envolvendo a criação de uma hash para ser utilizada como senha da assinatura, a qual pode ser enviada a conta desejada. Além disso, pode-se buscar detalhar a construção e funcionamento da aplicação web, implementando-se o processo de TDD em sua arquitetura.

ARCHITECTURAL PROPOSAL FOR DECENTRALIZED APPLICATIONS

Paulo Cesar Martins Citron[¶] Jorge Luis Boeira Bavaresco[‖]

28 de julho de 2022

Abstract

Web 3.0 and decentralized applications allow the creation of complex systems without the need for servers and managing authorities. This work aims to contribute to developing these applications through the proposition of architecture for its construction. By implementing the architecture proposed in a case study, an application for managing and signing digital contracts, it was possible to test the solution. The implementation results confirm that the architecture modelling is functional and support the view that building systems aiming at their autonomy and privacy protection is possible.

Keywords: Decentralized Applications. Blockchain. Ethereum. Web 3.0.

Referências

ANDERSON, C. Docker [software engineering]. *IEEE Software*, v. 32, n. 3, p. 102–c3, 2015. Citado na página 5.

ANWER, F. et al. Agile software development models tdd, fdd, dsdm, and crystal methods: A survey. *International Journal of Multidisciplinary Sciences and Engineering*, v. 8, n. 2, 3 2017. Citado na página 7.

CROSBY, M. et al. Blockchain technology: Beyond bitcoin. *AIR*, p. 6–19, 06 2016. Citado na página 3.

DOCKER. *Docker*. 2022. Disponível em: <https://www.docker.com/why-docker/>. Acesso em: 12 jul 2022. Citado na página 5.

[¶]paulo.citron@proton.me
[‖]jorgebavaresco@ifsul.edu.br

ETHEREUM. *Ethereum*. 2022. Disponível em: <https://ethereum.org>. Acesso em: 12 jul 2022. Citado na página 4.

ETHERS. *Ethers*. 2022. Disponível em: <https://docs.ethers.io>. Acesso em: 12 jul 2022. Citado na página 5.

GACKENHEIMER, C. What is react? *Apress*, 2015. Citado na página 4.

GREEN, H. *Introducing the DAO: The organisation that will kill corporations*. 2016. Disponível em: <https://www.cityam.com/introducing-the-dao-the-organisation-that-will-kill-corporations/>. Acesso em: 12 jul 2022. Citado na página 2.

METAMASK. *Metamask*. 2022. Disponível em: <https://metamask.io>. Acesso em: 12 jul 2022. Citado na página 4.

NATH, K.; DHAR, S.; BASISHTHA, S. Web 1.0 to web 3.0 - evolution of the web and its various challenges. In: . [S.l.: s.n.], 2014. p. 86–89. ISBN 978-1-4799-2995-5. Citado na página 2.

NETO, A. F. da S. *Web 3.0: O que é ela e por que é significativo?* 2022. Disponível em: <https://www.geeklando.com.br/o-que-e-web-3-0/>. Acesso em: 12 jul 2022. Citado na página 3.

NORTON, C. Blockchain: Everything you need to know about the technology behind bitcoin. *Pronoun*, 2017. Citado na página 4.

REACT. *React*. 2022. Disponível em: <https://reactjs.org>. Acesso em: 12 jul 2022. Citado na página 5.

SOLIDITY. *Solidity*. 2022. Disponível em: <https://soliditylang.org>. Acesso em: 12 jul 2022. Citado na página 4.

SZABO, N. Smart contracts: Building blocks for digital markets. *Extropy*, n. 16, 1996. Citado na página 3.

TRUFFLESUITE. *TruffleSuite*. 2022. Disponível em: <https://trufflesuite.com>. Acesso em: 12 jul 2022. Citado na página 4.

VITE. *Vite*. 2022. Disponível em: <https://vitejs.dev>. Acesso em: 12 jul 2022. Citado na página 5.

WOOD, G. *Why We Need Web 3.0*. 2018. Disponível em: <https://gavofyork.medium.com/why-we-need-web-3-0-5da4f2bf95ab>. Acesso em: 12 jul 2022. Citado 2 vezes nas páginas 1 e 2.