VINÍCIUS ANTÔNIO CARVALHO LEITE

UMA BIBLIOTECA PARA MANIPULAÇÃO DE BLOCKCHAIN

LAVRAS - MG

2023

VINÍCIUS ANTÔNIO CARVALHO LEITE

UMA BIBLIOTECA PARA MANIPULAÇÃO DE BLOCKCHAIN

Projeto Acadêmico apresentado à Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, para a obtenção do título de Bacharel.

Prof. Dr. Ramon Gomes Costa

Orientador

LAVRAS - MG

2023

VINÍCIUS ANTÔNIO CARVALHO LEITE

UMA BIBLIOTECA PARA MANIPULAÇÃO DE BLOCKCHAIN

Projeto Acadêmico apresentado à Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, para a obtenção do título de Bacharel.

APROVADA em 11 de Julho de 2023.

Prof. Dr. Ramon Gomes Costa	UFLA
Prof. Dr(a). Renata Teles Moreira	UFLA
Prof. Dr. Paulo Afonso Parreira Júnior	UFLA

Prof. Dr. Ramon Gomes Costa
Orientador

LAVRAS - MG
2023

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem por objetivo apresentar o projeto da biblioteca "VTypeChain", uma biblioteca para manipulação de blockchain desenvolvida utilizando NodeJs com Typescript.

A história da blockchain se inicia na década de 1990, mas sua relevância e aplicação em larga escala são recentes. O conceito de uma rede que pode armazenar transações com segurança e transparência tem suas origens em um artigo publicado por Stuart Haber e W. Scott Stornetta (HABER. . . , 1991), intitulado "*How to Time-Stamp a Digital Document*".

No artigo, Haber e Stornetta propuseram um sistema que usava funções criptográficas para tornar as assinaturas digitais à prova de falsificação e garantir a integridade dos dados. No entanto, eles reconheceram que seu sistema ainda exigia a confiança em uma autoridade central para registrar e manter o carimbo do tempo (*timestamp*).

A solução para esse problema veio em 2008, quando um indivíduo ou grupo de indivíduos sob o pseudônimo "Satoshi Nakamoto", lançou o *white paper* do Bitcoin (NAKAMOTO. . . , 2008), descrevendo um sistema de dinheiro digital descentralizado baseado em uma rede de nós que verificavam e registravam transações em um livro-razão público, conhecido como *blockchain*.

De acordo com o artigo "A História Da Tecnologia Blockchain: Conheça Sua Timeline" do blog 101 Blockchains (101. . . , 2018), a criação do Bitcoin, que é uma forma de dinheiro digital descentralizado e criptografado, ou seja, uma criptomoeda, e da *blockchain* trouxe uma nova abordagem para a gestão de dados e transações financeiras, removendo a necessidade de uma autoridade central para validar as transações, pois da maneira tradicional, há a necessidade de um intermediário, como as instituições financeiras, para se executar uma transação, já com a *blockchain* não há intermediários. Como resultado, a *blockchain* não apenas permitiu transações financeiras seguras e confiáveis, mas também trouxe a promessa de aplicativos descentralizados em uma variedade de setores. Desde então, a *blockchain* tem sido explorada e aplicada em uma variedade de áreas, desde pagamentos e finanças até identidade digital, votação eletrônica, cadeia de suprimentos e outras. Ademais, em 2013 surge a *blockchain* Etherium, desenvolvida por Vitalik Buterin, que foi um dos primeiros colaboradores do Bitcoin. A *blockchain* Etherium permite não apenas as transações, mas também a execução de contratos inteligentes, que são "contratos digitais" autoexecutáveis que facilitam, verificam e executam automaticamente acordos entre duas ou mais partes, com base em termos e condições pré-definidos, e são projetados para serem executados em uma *blockchain*. No entanto, muitas pessoas ainda não sabem como utilizar essa tecnologia em seus projetos. Neste

contexto, a criação de uma biblioteca de manipulação de *blockchain* surge como uma solução para tornar a tecnologia mais acessível e fácil de usar.

A criação de tal biblioteca é importante pois ela permite que os desenvolvedores criem aplicativos *blockchain* de forma mais eficiente e simplificada. Isso significa que mais pessoas podem começar a desenvolver projetos utilizando *blockchain* e expandir o uso da tecnologia para além das limitações atuais. Além disso, a biblioteca pode ser usada como base para outras soluções e projetos futuros, tornando-se um recurso para a comunidade de desenvolvedores.

Essa biblioteca possui diversas aplicações. Dentre elas, pode ser utilizada para criar aplicativos de gerenciamento de identidade, votação eletrônica, gerenciamento de ativos e muito mais. A linguagem Typescript, que amplia as funcionalidades do Javascript convencional, é altamente escalável e possui recursos de tipagem estática, tornando-a ótima escolha para esse tipo de projeto.

O público-alvo dessa biblioteca são principalmente os desenvolvedores interessados em criar aplicativos *blockchain* de forma mais fácil e rápida. Além disso, também pode ser útil para empresas e organizações que desejam implementar soluções *blockchain* em seus negócios.

Além deste capítulo introdutório será apresentado nos capítulos as tecnologias utilizadas no projeto, a descrição geral sobre *blockchain* e a estrutura da biblioteca desenvolvida, a implementação da biblioteca, a prova de conceito para colocar a biblioteca à prova e a conclusão.

2 TECNOLOGIAS UTILIZADAS

Neste capitulo são apresentadas as tecnologias utilizadas para o desenvolvimento da biblioteca.

2.1 Node.Js

O Node.js[1], ou simplesmente Node, é um ambiente de execução de JavaScript construído sobre o motor de JavaScript V8 do Google Chrome. Ele permite que os desenvolvedores escrevam código JavaScript do lado do servidor, em vez de apenas no lado do cliente.

O Node foi inicialmente desenvolvido em 2009 por Ryan Dahl e, desde então, tem se tornado cada vez mais popular como uma plataforma para desenvolvimento de aplicativos escaláveis de alta performance. Ademais, é adequado para aplicativos de rede que exigem I/O (Entrada e Saída) intensivo, como aplicativos de *chat*, plataformas de jogos *online*, aplicativos de colaboração em tempo real e aplicativos de *streaming* de mídia. Além disso, é usado por muitas empresas de tecnologia, incluindo Netflix, LinkedIn, Walmart e Uber.

Ele também possui um gerenciador de pacotes incorporado chamado npm, que é um repositório online de pacotes que podem ser instalados e gerenciados em um projeto node.

No projeto, o NodeJs foi utilizado para dar suporte à linguagem Typescript, e permitir sua execução do lado do servidor.

2.2 Typescript

Typescript[2] é um *superset* da linguagem de programação JavaScript, que adiciona recursos avançados à linguagem, como tipagem estática, *interfaces*, classes, *enumerates*, e outras construções de linguagem que são comuns em linguagens de programação orientadas a objetos.

Ele foi desenvolvido pela Microsoft e lançado em 2012 visando principalmente a criação de aplicativos maiores e mais complexos. O TypeScript é um software livre (de código aberto), e pode ser instalado em diferentes sistemas operacionais. Uma de suas vantagens, sua tipagem estática, que permite aos desenvolvedores detectar erros de tipagem antes mesmo de executar o código, o que pode melhorar a qualidade do projeto.

[1] https://nodejs.org/en
[2] https://www.typescriptlang.org/

O TypeScript é compatível com as bibliotecas e *frameworks* JavaScript existentes, tornando-se uma escolha para projetos que utilizam essas tecnologias. Ele é suportado por diferentes editores de texto e ambientes de desenvolvimento, como o Visual Studio Code e o WebStorm, que fornecem ferramentas de edição de código avançadas, como completar código, refatoração de código, depurar e outras.

Ele é usado em projetos, desde aplicativos para *desktop*, aplicativos móveis até aplicações Web em larga escala, como Angular, React, Vue.js e outras. É adequado para aplicativos grandes e complexos que exigem uma grande equipe de desenvolvedores, pois ajuda a evitar erros e a manter o código organizado.

No projeto, o Typescript é utilizado na implementação da lógica de negócio referente à *blockchain*, aos blocos e às transações, bem como as configurações necessárias à biblioteca.

2.3 Microbundle

O Microbundle[3] é uma ferramenta de empacotamento (*bundling*) de módulos JavaScript para a criação de pacotes (*packages*) otimizados para produção. Ele é uma alternativa leve e simples a outras ferramentas de empacotamento, como o Webpack e o Rollup, que são mais complexas e podem exigir configuração adicional.

O Microbundle utiliza a biblioteca Rollup como base, mas adiciona várias melhorias e convenções para tornar o empacotamento de módulos JavaScript mais fácil e rápido. Ela suporta módulos ECMAScript (ES6) e CommonJS, bem como outras construções de linguagem, como importação de arquivos CSS e JSON.

Seu uso é adequado para a criação de bibliotecas JavaScript que serão utilizadas por outros desenvolvedores em seus projetos. Ela cria um pacote otimizado para produção, que inclui todas as dependências necessárias, eliminando a necessidade de instalar dependências em um projeto cliente. Além disso, a Microbundle inclui suporte para o uso de TypeScript e outras ferramentas de construção, como Babel.

No Projeto, o Microbundle foi utilizado para fazer o *bundling* da aplicação e converter todo o código Typescript em Javascript.

[3] https://www.npmjs.com/package/microbundle?activeTab=readme

2.4 CryptoJs

CryptoJs[4] é uma biblioteca com código aberto de criptografia de dados escrita em JavaScript, que fornece uma gama de algoritmos de criptografia para proteger informações confidenciais. Ela é utilizada principalmente em aplicações Web para proteger dados que são transmitidos pela internet ou armazenados em bancos de dados.

A CryptoJS fornece uma série de funções criptográficas para diferentes algoritmos, como AES, DES, HMAC, SHA-1 e SHA-256. Essas funções podem ser usadas para criptografar e descriptografar dados, gerar chaves de criptografia além de calcular hashes de mensagem e objetos.

Além disso, a biblioteca oferece recursos avançados, como a capacidade de trabalhar com dados em diferentes formatos, como texto, arquivos, objetos e ArrayBuffer, que são utilizados para armazenar dados binários. E pode ser integrada em projetos JavaScript ou TypeScript, e amplamente utilizada em projetos que precisam de segurança de dados, como aplicativos bancários, sistemas de pagamento *online*, sistemas de gerenciamento de senhas, e muito mais.

Uma das principais vantagens da CryptoJS é sua simplicidade de uso, uma vez que o desenvolvedor precisa apenas incluir a biblioteca em seu projeto e chamar as funções correspondentes para criptografar ou decriptografar os dados. Isso torna a biblioteca uma escolha para desenvolvedores que precisam adicionar segurança aos seus projetos sem precisar de um conhecimento profundo em criptografia.

No projeto, a CryptoJs foi utilizada para gerar a criptografia dos blocos da *blockchain* e das transações, aplicando a função criptográfica SHA256.

2.5 Elliptic

Elliptic[5] é uma biblioteca JavaScript de criptografia que implementa curvas elípticas para fornecer recursos de criptografia, incluindo geração de chaves, assinaturas digitais, criptografia de dados e outros. A biblioteca é usada principalmente para proteger dados confidenciais em aplicativos da Web.

Curvas elípticas são uma área da matemática que trata de curvas definidas por equações polinomiais e suas propriedades algébricas. Elas têm sido usadas para criptografia há várias décadas,

[4] https://www.npmjs.com/package/crypto-js?activeTab=readme
[5] https://www.npmjs.com/package/elliptic?activeTab=readme

mas recentemente se tornaram populares devido a sua eficiência e segurança em relação a outros algoritmos criptográficos.

Elliptic usa curvas elípticas para implementar a criptografia de chave pública, que é usada para gerar pares de chaves pública e privada para proteger dados. A biblioteca suporta vários algoritmos de curva elíptica, como a curva secp256k1, que é usada pelo Bitcoin e outras criptomoedas para gerar chaves públicas e privadas para carteiras digitais.

A biblioteca oferece suporte para assinaturas digitais usando o algoritmo ECDSA (*Elliptic Curve Digital Signature Algorithm*), que é usado para verificar a autenticidade de dados assinados digitalmente. Por fim, pode ser usada para criptografia de dados, que envolve a codificação de dados em um formato seguro e irreversível.

No projeto, a Elliptic foi utilizada para implementar o mecanismo de assinatura digital nas transações da *blockchain*.

3 DESCRIÇÃO GERAL DO TEMA

O projeto da biblioteca para manipulação de *blockchain* surgiu a partir do momento em que um colega de trabalho do autor foi contratado por outra empresa para trabalhar com essa tecnologia, despertando a curiosidade do autor deste trabalho em relação ao universo das *blockchains*. Desse modo, inicialmente foi procurado vídeos no Youtube (YOUTUBE, 2023) que abordavam conceitos básicos sobre o assunto, encontrando um tutorial de implementação simplificada de uma *blockchain* para criptomoedas. Acompanhou-se os tutoriais e foi desenvolvida a primeira *blockchain*, porém se percebeu a necessidade de aprofundamento, uma vez que a implementação era limitada.

Ampliou-se a pesquisa para outros vídeos e repositórios do GitHub (GITHUB, 2023), encontrando conteúdos úteis, porém a maioria não fornecia as informações desejadas. Com base nos conhecimentos adquiridos, consolidou-se a ideia de criar uma biblioteca para gerenciar *blockchains*, com o propósito de auxiliar desenvolvedores iniciantes e permitir a contribuição da comunidade, tornando-a uma biblioteca de código aberto.

No segundo trimestre de 2022, começou-se a amadurecer a ideia e planejar a estrutura da biblioteca, assim como as tecnologias a serem utilizadas. Ademais, foi delineado um esboço inicial que abrangia apenas o gerenciamento de *blockchains* para criptomoedas. Nomeou-se o projeto como "Projeto VTypeChain", derivado de uma combinação de referências, incluindo o nome "Vinícius", a linguagem "Typescript"e a tecnologia "*Blockchain*".

3.1 Estrutura da Blockchain

A estrutura de uma *blockchain* é formada basicamente por três componentes, sendo eles a *blockchain* os blocos e as transações, o pleno entendimento sobre esses três componentes é importante para a compreensão do funcionamento e implementação de uma *blockchain*. Portanto, antes de seguir para a estrutura da biblioteca, primeiro será apresentado uma explicação acerca desses três componentes.

O bloco é uma estrutura de dados fundamental em uma *blockchain*, que contém um registro de transações recentes, bem como outras informações relevantes para a rede. Cada bloco é conectado a um bloco anterior na cadeia, criando assim uma cadeia de blocos.

Um bloco típico contém as seguintes informações:

- Hash do bloco anterior: o que garante a integridade da cadeia e impede que blocos antigos sejam alterados;

- Timestamp: a data e hora em que o bloco foi criado;
- Nonce: um número aleatório usado para criar o *hash* do bloco;
- Lista de transações: uma lista de transações incluídas no bloco;
- Hash do bloco: um *hash* único gerado a partir de todas as informações contidas no bloco, incluindo o *hash* do bloco anterior, o *timestamp*, o *nonce* e as transações;

O processo de criação de um novo bloco em uma *blockchain* é chamado de mineração. Os mineradores competem para resolver um problema matemático complexo para criar um novo bloco e adicioná-lo à cadeia. Esse processo exige uma grande quantidade de poder computacional e, uma vez que o problema é resolvido, o bloco é adicionado à *blockchain* e a recompensa é recebida pelo minerador bem-sucedido.

A transação é um registro de transferência de dados ou ativos digitais entre participantes de uma rede de *blockchain*. Pode-se pensar como uma operação financeira ou uma interação que ocorre na rede. Essa transação é registrada em um bloco da cadeia de blocos e validada pelos mineradores, que garantem que a transação seja autêntica.

As transações são essenciais para a *blockchain*, pois permitem que as criptomoedas sejam transferidas entre as partes sem a necessidade de intermediários, como bancos ou outras instituições financeiras. Além disso, as transações na *blockchain* são transparentes e imutáveis, o que significa que uma vez que uma transação é registrada na *blockchain*, ela não pode ser alterada ou excluída.

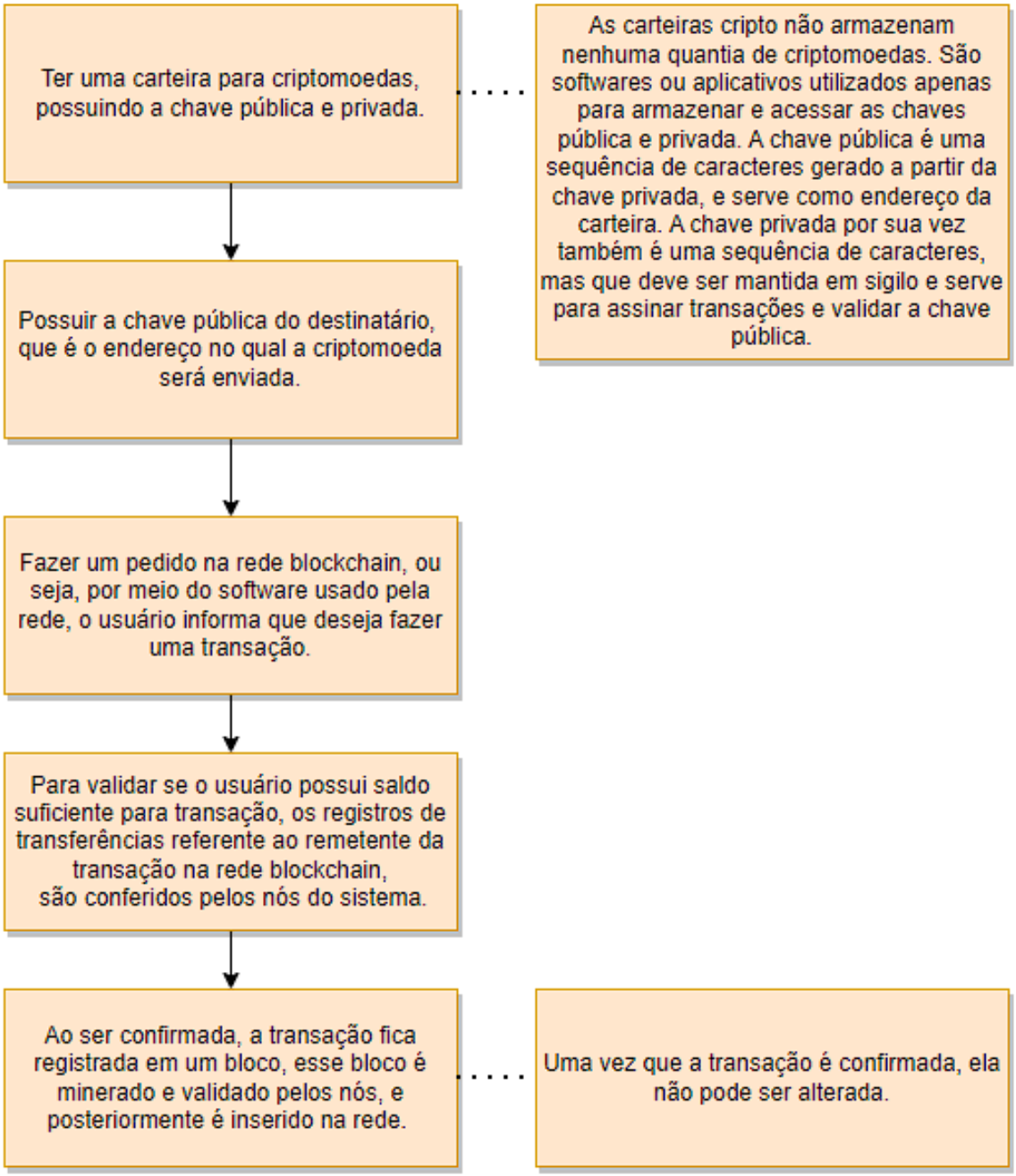
Uma transação na *blockchain* pode ser composta por várias informações, incluindo:

- Endereço do remetente: o endereço da carteira digital que envia a criptomoeda;
- Endereço do destinatário: o endereço da carteira digital que recebe a criptomoeda;
- Quantidade enviada: a quantidade de criptomoeda que está sendo enviada;
- Timestamp: a data e hora em que o bloco foi criado;
- Assinatura digital: uma assinatura digital que garante a autenticidade da transação;
- Taxa de transação: recompensa para os mineradores. Algumas implementações podem conter essa taxa na própria transação, e outras costumam aplicar a recompensa pela própria *blockchain*;

Uma vez que uma transação é criada e enviada para a rede, os mineradores trabalham para validar a transação e adicioná-la a um bloco. Isso envolve verificar a autenticidade da transação e confirmar que o remetente tem fundos suficientes para enviar a criptomoeda ou *token*. Quando a transação é confirmada e adicionada a um bloco, ela se torna uma parte permanente do registro na *blockchain*.

A seguir, é mostrado na Figura 3.1, o fluxo de uma transação de criptomoeda.

Figura 3.1 – Fluxo de uma Transação de Criptomoeda



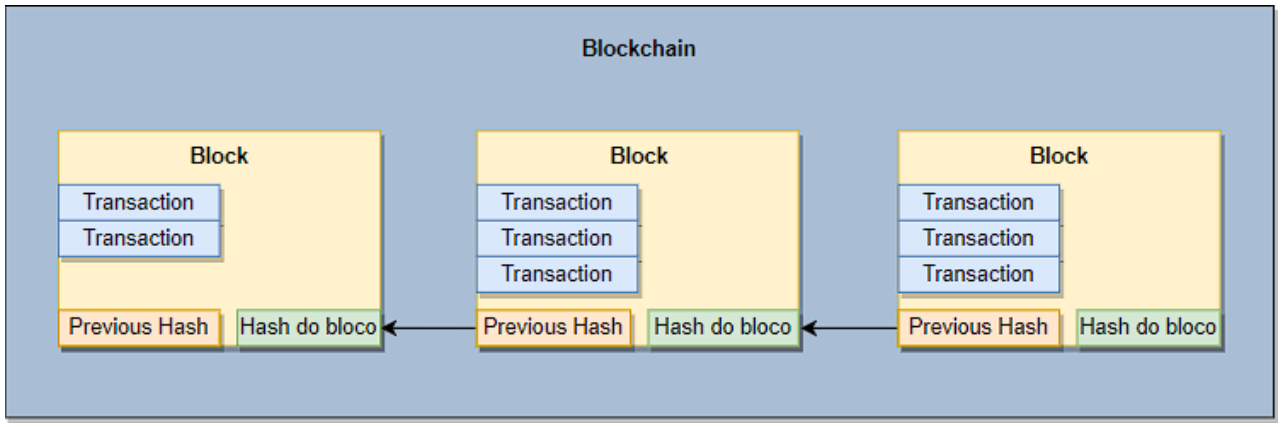
Fonte: Do Autor

A Blockchain é um componente fundamental em uma *blockchain*, pois é responsável por realizar a orquestração de todos os blocos que compõem a cadeia de blocos. Nesse sentido, sua implementação se faz necessário para criar, validar e adicionar novos blocos a essa lista.

A seguir, na Figura 3.2, é apresentada a estrutura de uma *blockchain*, onde apresenta que a *blockchain* armazena e gerencia os blocos, que por sua vez armazenam as transações. Para encadear

os blocos, usa-se o *hash* do bloco anterior, pois assim é possível identificar sua ordem perante a cadeia de blocos.

Figura 3.2 – Estrutura da Blockchain



Fonte: Do Autor

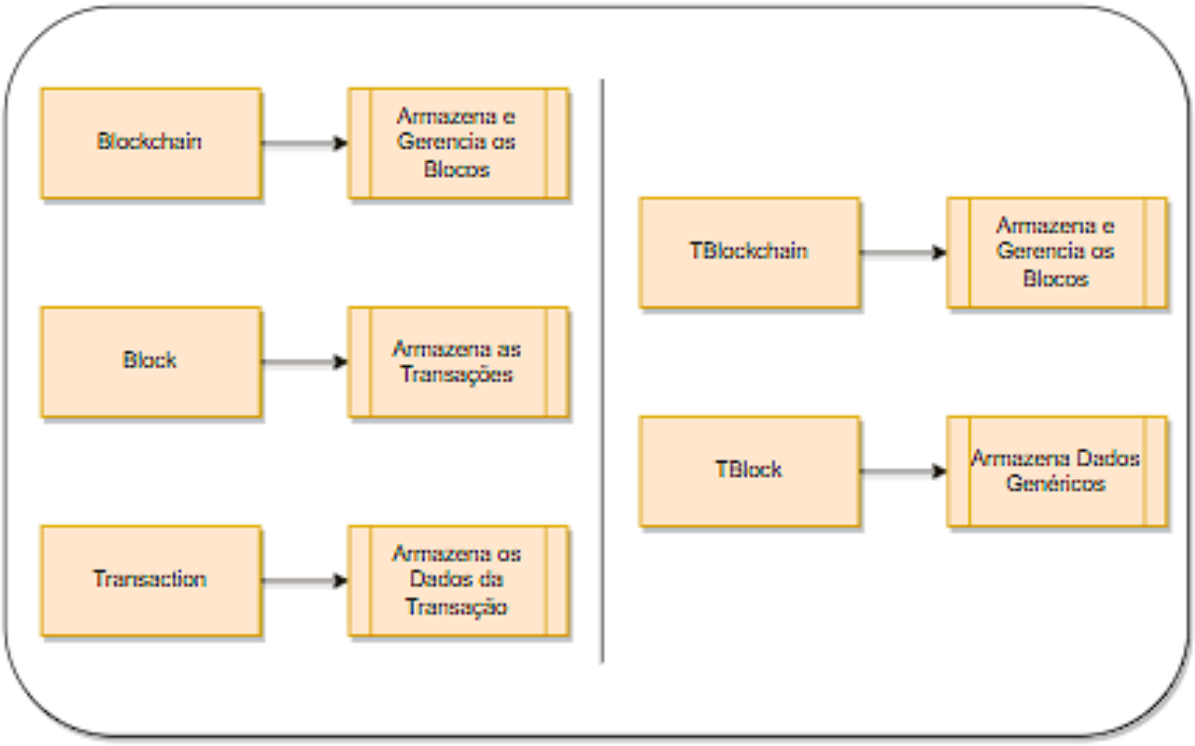
Na classe Blockchain é importante haver algumas propriedades para a manipulação da cadeia de blocos. Sendo elas:

- Cadeia de Blocos: uma lista que contém todos os blocos da cadeia;
- Dificuldade: a dificuldade da prova de trabalho (PoW) usada para minerar novos blocos;
- Transações pendentes: uma lista de transações que ainda não foram adicionadas a um bloco;
- Recompensa por mineração: a recompensa dada para o minerador ao minerar com sucesso, mas isso pode variar dependendo da implementação e de acordo com o objetivo da criptomoeda.

3.2 Estrutura da Biblioteca

A seguir, na Figura 3.3, pode ser visto o planejamento das classes, contendo seus respectivos nomes e responsabilidades.

Figura 3.3 – Classes do Projeto



Fonte: Do Autor

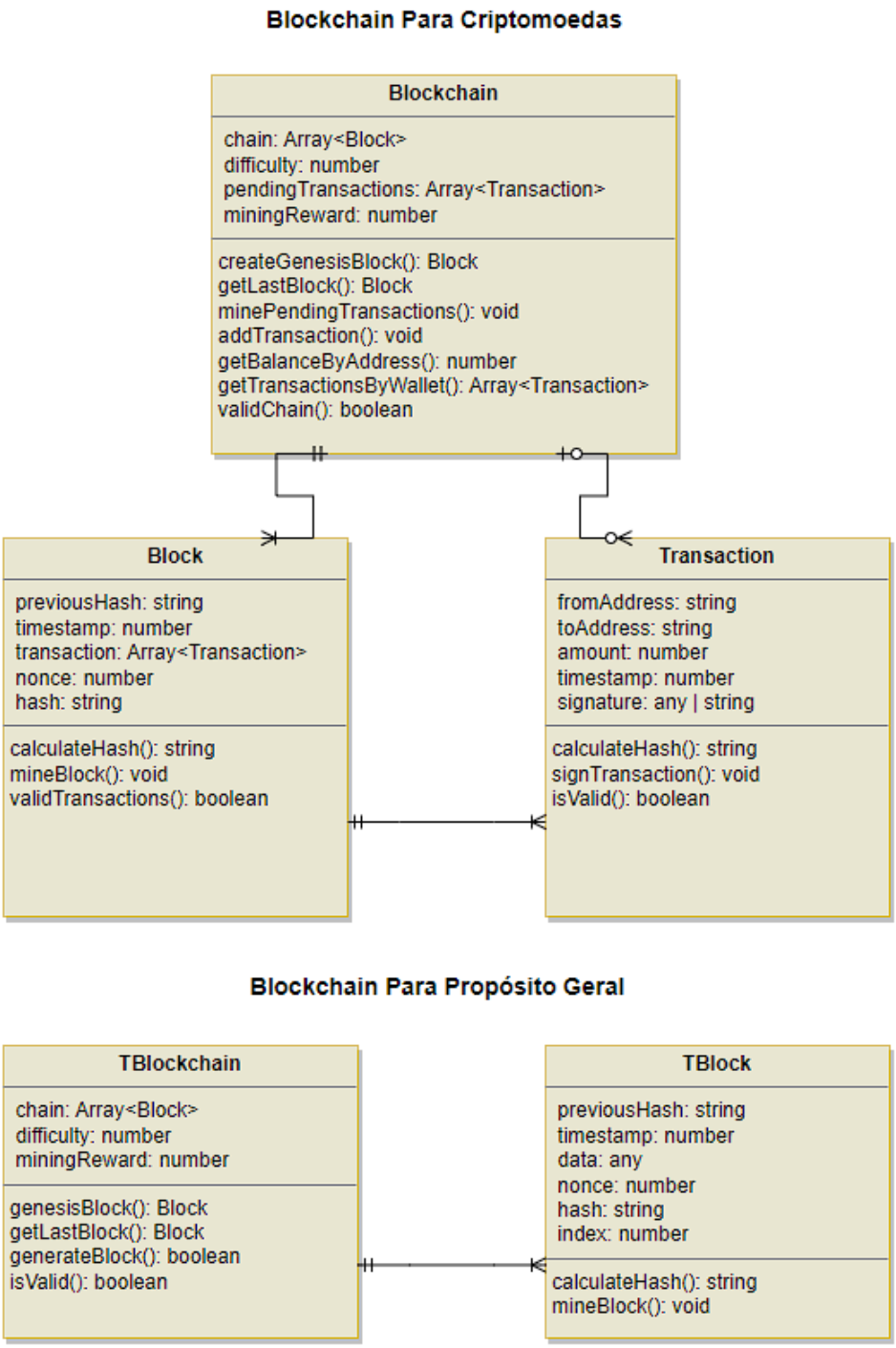
Desta forma, após a definição e implementação, o projeto obteve dois tipos de *blockchain*, a primeira para criptomoedas e a segunda para propósitos gerais, como apresentado na Figura 3.4.

A implementação segregando esses dois tipos de *blockchain* é para que, no futuro, seja mais fácil o incremento de novas funcionalidades específicas para cada tipo de uso. Desse modo, cada módulo pode evoluir de forma independente.

Para melhor entendimento, serão apresentados dois exemplos de uso da biblioteca, um para cada tipo de *blockchain*. Um exemplo para o uso da *blockchain* para criptomoedas pode ser: um mecanismo de moeda digital para uma loja virtual, em que, ao usuário comprar algum produto recebe uma quantia dessa moeda.

Um exemplo para o uso da *blockchain* para propósito geral pode ser: um sistema de rastreamento de etapas de uma cadeia de suprimentos. À medida que o produto avança pela cadeia de suprimentos, as etapas são registradas na *blockchain*. Estas, podem incluir informações como transporte, embalagem, armazenamento e outros processos no qual o produto passa. Desse modo, cada participante da cadeia de suprimentos pode adicionar informações relevantes à *blockchain*, garantindo a transparência.

Figura 3.4 – Classes do Projeto



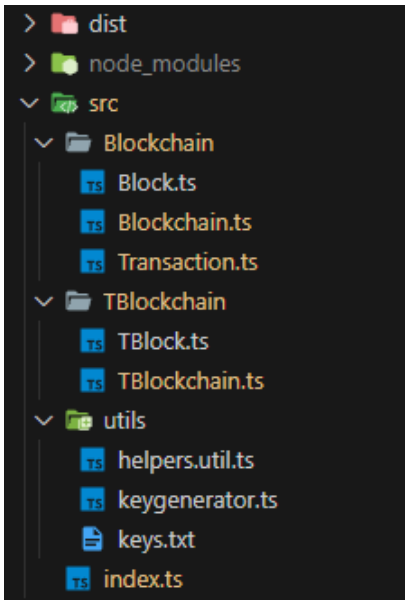
Fonte: Do Autor

Ademais, a implementação foi feita usando a linguagem Typescript, para garantir a tipagem das propriedades das classes e os retornos dos métodos. Desse modo, é possível manter a verificação de tipos e garantir que serão obedecidos por todas as classes.

A seguir, pode ser observado a estrutura de diretórios do projeto na Figura 3.5:

- Diretório "dist": utilizado para armazenar os arquivos do empacotamento da biblioteca. Este diretório se encontra na raiz do projeto;
- Diretório "src": utilizado para armazenar a implementação das classes do projeto e funções auxiliares. Este diretório se encontra na raiz do projeto;
- Diretório "Blockchain": utilizado para armazenar a implementação da *blockchain* para criptomoedas. Este diretório se encontra dentro de "src".
- Diretório "TBlockchain": utilizado para armazenar a implementação da *blockchain* para propósito geral. Este diretório se encontra dentro de "src".
- Diretório "Utils": utilizado para armazenar as funções auxiliares do projeto. Este diretório se encontra dentro de "src".

Figura 3.5 – Estrutura de Diretórios



Fonte: Do Autor

4 IMPLEMENTAÇÃO

Neste capítulo são apresentadas as características da implementação de uma *blockchain*, bem como a estruturação da biblioteca. Para isso, é necessário entender as características de uma *blockchain* e o seu funcionamento. Com isso, as explicações serão baseadas nas partes de uma *blockchain*. Mas antes, é válido ressaltar que a biblioteca desenvolvida possui duas implementações de *blockchain*: uma mais simples, capaz de armazenar qualquer tipo de dados, sendo a melhor escolha para implementações para cadeia de suprimentos ou apenas armazenamento de informações; e outra voltada para criptomoedas, sendo a que vamos abordar neste trabalho, pois com ela poderemos ter um melhor entendimento sobre o tema, devido a sua implementação abordar mais conceitos.

4.1 O Bloco

No Código 4.1 é apresentada a implementação da classe Block e logo a seguir, sua explicação.

Código 4.1 – Exemplo de Implementação do Bloco.

```
1  import { Transaction } from './Transaction';
2  import sha256 from 'crypto-js/sha256';
3
4  export class Block {
5    previousHash: string;
6    timestamp: number;
7    transactions: Transaction[];
8    nonce = 0;
9    hash = this.calculateHash();
10
11   constructor(timestamp: number, transactions: Transaction[],
       previousHash: string = '') {
12     this.previousHash = previousHash;
13     this.timestamp = timestamp;
14     this.transactions = transactions;
15     this.nonce = 0;
16     this.hash = this.calculateHash();
```

```
17    }
18
19    calculateHash(nonce = this.nonce): string {
20      return sha256(this.previousHash + this.timestamp + JSON.stringify
          (this.transactions) + nonce).toString();
21    }
22
23    mineBlock(difficulty: number): void {
24      const target = Array(difficulty + 1).join('0');
25      let nonce = 0;
26      let hash = this.calculateHash();
27
28      while (hash.substring(0, difficulty) !== target) {
29        nonce++;
30        hash = this.calculateHash(nonce);
31      }
32
33      this.nonce = nonce;
34      this.hash = hash;
35      console.log(`Block mined: ${this.hash}`);
36    }
37
38    validTransactions(): boolean {
39      for (const transaction of this.transactions) {
40        if (!transaction.isValid()) {
41          return false;
42        }
43      }
44      return true;
45    }
46  }
```

Fonte: do Autor

Como apresentado no Código 4.1, o bloco é uma estrutura que armazena as informações que a *blockchain* irá conter, que neste caso são as transações. Outro ponto importante é o método de mineração do bloco, que é o mecânismo de consenso mais simples para *blockchains*, chamado de *Proof-of-Work* (PoW). Ele é baseado em um processo de mineração que envolve a resolução de um problema matemático complexo, a fim de validar a transação ou gerar um novo bloco.

O PoW é um processo em que os mineradores competem entre si para resolver o problema matemático e encontrar uma solução válida. O primeiro minerador a encontrar a solução válida é recompensado com uma quantidade de criptomoeda como incentivo. Com isso, adiante será apresentado a explicação do algoritmo implementado.

O parâmetro difficulty (linha 23), é a dificuldade definida para o bloco. Quanto maior a dificuldade, mais difícil é encontrar um *hash* válido, o que requer mais poder computacional.

A variável target (linha 24), é uma *string* de zeros usada para verificar se o *hash* do bloco encontrado possui a dificuldade desejada. A quantidade de zeros na *string* target é determinada pelo parâmetro difficulty.

A variável nonce (linha 29), é um número inteiro que é incrementado a cada iteração do *loop* while até que um hash válido seja encontrado.

A variável hash (linha 26), é inicializada com o *hash* do bloco atual, que é obtido através do método calculateHash da classe.

O *loop* while (linha 28) é executado até que o *hash* encontrado comece com a quantidade de zeros definidos no parâmetro difficulty, ou seja, até que o *hash* encontrado atenda aos critérios de validade. Dentro do *loop*, o nonce é incrementado a cada iteração e o *hash* é recalculado com o novo nonce através do método calculateHash.

Uma vez encontrado um *hash* válido que atenda aos critérios de dificuldade, o valor do nonce e o hash válido são atribuídos às variáveis nonce e hash da classe, respectivamente. Em seguida, uma mensagem é exibida informando que o bloco foi minerado com sucesso, exibindo o *hash* encontrado.

4.2 A Transação

No Código 4.2 é apresentada a implementação da classe Transaction e logo a seguir, sua explicação.

Código 4.2 – Exemplo de Implementação da Transação.

```
1  import sha256 from 'crypto-js/sha256';
2  const EC = require('elliptic').ec
3  const ec = new EC('secp256k1')
4
5  export class Transaction {
6    fromAddress: string;
7    toAddress: string;
8    amount: number;
9    timestamp: number;
10   signature: any = null;
11
12   constructor(fromAddress: string, toAddress: string, amount: number)
         {
13     this.fromAddress = fromAddress;
14     this.toAddress = toAddress;
15     this.amount = amount;
16     this.timestamp = Date.now();
17   }
18
19   calculateHash(): string {
20     return sha256(this.fromAddress + this.toAddress + this.amount +
           this.timestamp).toString();
21   }
22
23   signTransaction(signingKey: any): void {
24     if (signingKey.getPublic('hex') !== this.fromAddress)
25       throw new Error('Cannot sign transactions of another wallet.');
26
27     const hashTx = this.calculateHash();
```

```
28      const sig = signingKey.sign(hashTx, 'base64');
29
30      this.signature = sig.toDER('hex');
31    }
32
33    isValid(): boolean {
34      if (!this.signature || this.signature.length === 0)
35        return false
36
37      if (!this.fromAddress || !this.toAddress)
38        return false
39
40      if (this.amount <= 0)
41        return false;
42
43      if (this.fromAddress === null) return true;
44
45      const publicKey = ec.keyFromPublic(this.fromAddress, 'hex');
46      return publicKey.verify(this.calculateHash(), this.signature);
47    }
48  }
```

Fonte: do Autor

O Código 4.2 apresenta uma implementação da transação na *blockchain*. Assim, seu funcionamento será descrito a seguir.

A classe Transaction tem cinco propriedades: fromAddress, toAddress, amount e timestamp e signature (linhas de 6 a 10). A partir dessas propriedades, é possível criar uma transação entre dois endereços de carteira e especificar uma quantidade de criptomoedas que serão transferidas.

O método calculateHash (linha 19), é utilizado para calcular o *hash* da transação, utilizando a função sha256 do módulo crypto-js. O *hash* é uma representação única da transação que será adicionada ao bloco.

O método signTransaction (linha 29) é utilizado para assinar a transação com a chave privada da carteira que está enviando as criptomoedas. Ele verifica na linha 24 se a chave privada corresponde à chave pública da carteira de origem e, em seguida, nas linhas de 27 a 30, gera a assinatura para a transação utilizando o método sign da biblioteca elliptic.

O método isValid (linha 33), é responsável por verificar se uma transação é válida ou não. A primeira verificação é se a transação possui uma assinatura, o que garante que a transação foi autorizada pelo remetente. Se não houver uma assinatura ou ela tiver comprimento zero, a transação é considerada inválida. A segunda verificação é se os endereços do remetente e do destinatário estão presentes. Se algum dos endereços não estiver presente, a transação é considerada inválida. A terceira verificação é se o valor da transação é maior que zero. Se o valor for menor ou igual a zero, a transação é considerada inválida. Por fim, a quarta verificação é se a transação foi assinada com a chave pública do remetente, garantindo que apenas o remetente possa autorizar a transação. Se a chave pública do remetente não for encontrada ou se a assinatura não for válida, a transação é considerada inválida.

Desse modo, obtém-se uma implementação simples e eficaz de uma transação com assinatura digital que é fundamental para garantir a autenticidade e a integridade da transação na *blockchain*. A assinatura digital é uma técnica criptográfica que utiliza chaves públicas e privadas para garantir que a transação é autêntica e que não foi alterada durante o seu envio e armazenamento na rede *blockchain*. Ela é gerada a partir da chave privada da carteira digital do remetente e é verificada pela chave pública associada à carteira. Dessa forma, apenas o remetente da transação pode assiná-la e a transação pode ser verificada por qualquer usuário da rede *blockchain*.

4.3 A Blockchain

A seguir, no Código 4.3 é apresentada a implementação da classe Blockchain e logo a seguir, sua explicação.

Código 4.3 – Exemplo de Implementação da Blockchain.

```
1  import { Block } from "./Block";
2  import { Transaction } from "./Transaction";
3
4  export class Blockchain {
5    chain: Block[];
6    difficulty: number;
7    pendingTransactions: Transaction[];
8    miningReward: number;
9
10   constructor(difficult: number = 4) {
11     this.chain = [this.createGenesisBlock()];
12     this.difficulty = difficult;
13     this.pendingTransactions = new Array<any>();
14     this.miningReward = 100;
15   }
16
17   createGenesisBlock(): Block {
18     return new Block(Date.now(), [], '0');
19   }
20
21   getLastBlock(): Block {
22     return this.chain[this.chain.length - 1];
23   }
24
25   minePendingTransactions(miningRewardAddress: string): void {
26     const rewardTr = new Transaction(
27       null,
28       miningRewardAddress,
29       this.miningReward
```

```
30      );
31      this.pendingTransactions.push(rewardTr);
32
33      const block = new Block(
34        Date.now(),
35        this.pendingTransactions,
36        this.getLastBlock().hash
37      );
38      block.mineBlock(this.difficulty);
39
40      this.chain.push(block);
41
42      this.pendingTransactions = [];
43    }
44
45    addTransaction(transaction: Transaction): void {
46      if (!transaction.isValid())
47        throw new Error('Cannot add invalid transaction.');
48
49      const walletBalance = this.getBalanceByAddress(transaction.
          fromAddress);
50      if (walletBalance < transaction.amount)
51        throw new Error('Balance is not enough.');
52
53      const pendingTrForWallet = this.pendingTransactions.filter(
54        tr => tr.fromAddress === transaction.fromAddress
55      );
56
57      if (pendingTrForWallet.length > 0) {
58        const totalPendingAmount = pendingTrForWallet
59          .map(tr => tr.amount)
60          .reduce((previous, current) => previous + current);
61
62        const totalAmount = totalPendingAmount + transaction.amount;
```

```
63        if (totalAmount > walletBalance)
64          throw new Error('Pending transactions for this wallet is
                higher than its balance.');
65      }
66
67      this.pendingTransactions.push(transaction);
68    }
69
70    getBalanceByAddress(address: string): number {
71      let balance = 0;
72
73      for (const block of this.chain) {
74        for (const transaction of block.transactions) {
75          if (transaction.fromAddress === address) {
76            balance -= transaction.amount;
77          }
78
79          if (transaction.toAddress === address) {
80            balance += transaction.amount;
81          }
82        }
83      }
84
85      return balance;
86    }
87
88    getTransactionsByWallet(address: string): Transaction[] {
89      const transactions = [];
90
91      for (const block of this.chain) {
92        for (const tr of block.transactions) {
93          if (tr.fromAddress === address || tr.toAddress === address) {
94            transactions.push(tr);
95          }
```

```
96          }
97        }
98
99        return transactions;
100     }
101
102     validChain(): boolean {
103       const genesisBlock = JSON.stringify(this.createGenesisBlock());
104
105       if (genesisBlock !== JSON.stringify(this.chain[0]))
106         return false;
107
108       for (let i = 1; i < this.chain.length; i++) {
109         const currentBlock = this.chain[i];
110         const previousBlock = this.chain[i - 1];
111
112         if (previousBlock.hash !== currentBlock.previousHash)
113           return false;
114
115         if (!currentBlock.validTransactions())
116           return false;
117
118         if (currentBlock.hash !== currentBlock.calculateHash())
119           return false;
120       }
121
122       return true;
123     }
124 }
```

Fonte: do Autor

O construtor da classe Blockchain (linha 10) do Código 4.3, cria uma lista com um bloco inicial, chamado de bloco gênesis, e define valores padrão para difficulty, pendingTransactions e miningReward.

O método createGenesisBlock (linha 17) do Código 4.3, cria um bloco vazio que é adicionado à cadeia de blocos, esse bloco é chamado de "Bloco Gênesis", por se tratar do primeiro bloco da cadeia.

O método getLastBlock (linha 21) do Código 4.3, retorna o último bloco adicionado à cadeia.

O método minePendingTransactions (linha 25) do Código 4.3, adiciona uma transação de recompensa para o minerador que minerou o próximo bloco (linhas 26 a 30), cria um novo bloco com as transações pendentes (linhas 33 a 38), adiciona o bloco à cadeia (linha 40), e por fim, deixa vazio a lista de transações pendentes.

O método addTransaction (linha 45) do Código 4.3, adiciona uma nova transação à lista de transações pendentes (linha 67), mas antes, ele verifica se a transação é válida (linhas 46 a 55), e se o saldo da carteira é suficiente para realizar a transação (linhas 57 a 65).

O método getBalanceByAddress (linha 70) do Código 4.3, calcula e retorna o saldo da carteira de um determinado endereço consultando todas as transações que afetam o endereço.

O método getTransactionsByWallet (linha 88) do Código 4.3, retorna todas as transações associadas a um determinado endereço de carteira.

O método validChain (linha 102) do Código 4.3, verifica se cada bloco da cadeia é válido. A validação inclui verificar se o bloco gênesis não foi alterado (linha 105), verificar se o *hash* do bloco anterior corresponde ao *hash* armazenado no bloco atual (linha 112), verificar se as transações contidas no bloco são válidas (linha 115) e se o *hash* do bloco está correto (linha 118).

É importante salientar que nessa implementação, a distribuição e operação em nós da *blockchain* fica a cargo do utilizador da biblioteca, pois desse modo, sua utilização se torna mais abrangente para diferentes tipos de uso.

5 PROVA DE CONCEITO

A prova de conceito consiste na implementação de uma aplicação *Web* que forneça uma interface gráfica para o usuário interagir com a *blockchain* desenvolvida, podendo criar transações, minerar blocos, verificar detalhes da carteira virtual e editar as suas configurações. Desse modo, certificando que a biblioteca cumpra com os requisitos.

5.1 Utilização da Biblioteca

Para instalar a biblioteca em um projeto, basta executar um único comando no terminal, sendo ele o ‘npm install github:Viniciud/vtype-chain‘, e toda a biblioteca será baixada direto do repositório do Github.

A bilbioteca exporta cinco classes para utilização das funcionalidades, sendo elas:

- VTypechain: uma *blockchain* para aplicações de armazenamento;
- VTypechainBlock: o tipo de bloco manipulado pela VTypechain, capaz de armazenar qualquer tipo de informação;
- VTypecoin: uma *blockchain* para criptomoedas.
- VTypecoinBlock: o tipo de bloco suportado pela VTypecoin.
- VTypeTransaction: o tipo de transação suportado e armazenado pelo VTypecoinBlock.

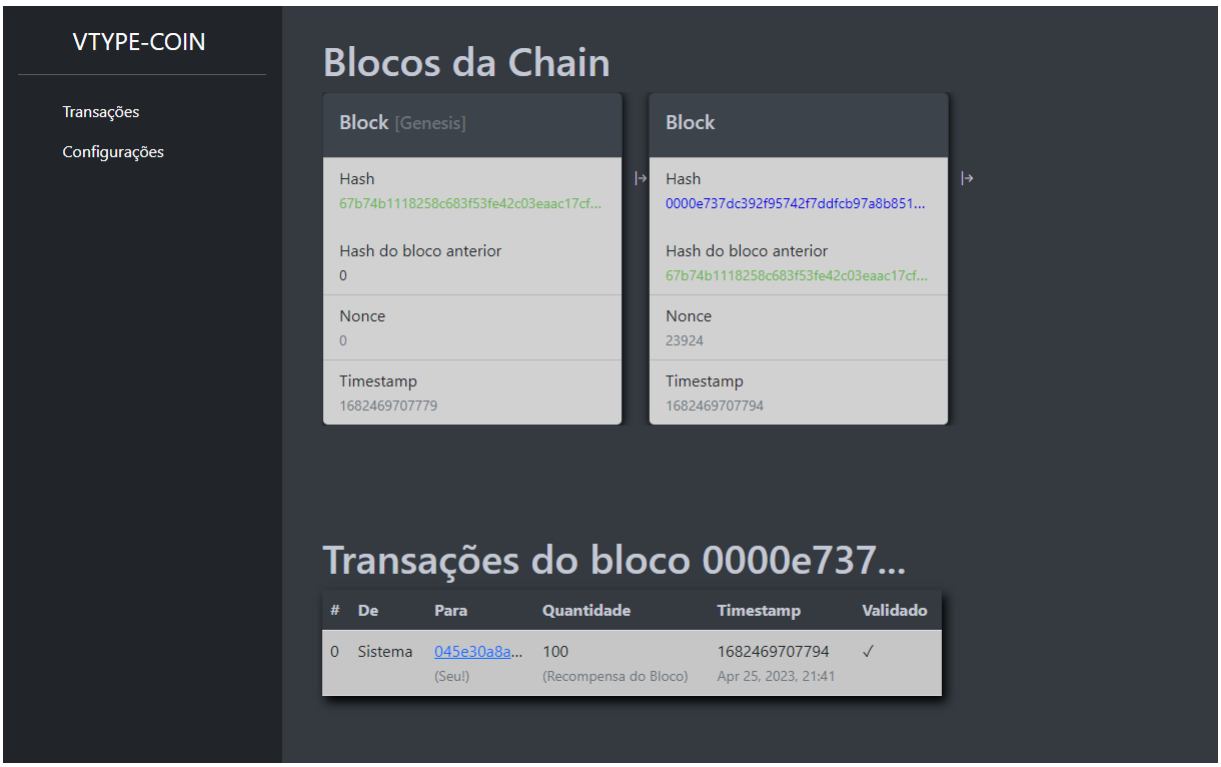
Para essa prova de conceito, é utilizada a VTypecoin, que utiliza a implementação apresentada no capítulo anterior.

5.2 A Aplicação da Biblioteca

Para criar uma aplicação *web* que comprove o funcionamento da biblioteca, foi necessário implementar algumas telas que apresentassem o funcionamento da *blockchain*. Para isso, foi utilizado o *framework* de desenvolvimento de interfaces *web* Angular (ANGULAR, 2023), por ser uma ferramenta de fácil utilização, e que já tenho certa afinidade.

Para a tela inicial, é apresentada uma interface onde o usuário tem a sua disposição as opções de "Transações"e "Configurações", além de apresentar os blocos da *blockchain* juntamente com as transações dentro do bloco selecionado. A Figura 5.1 apresenta a tela inicial com as transações do segundo bloco.

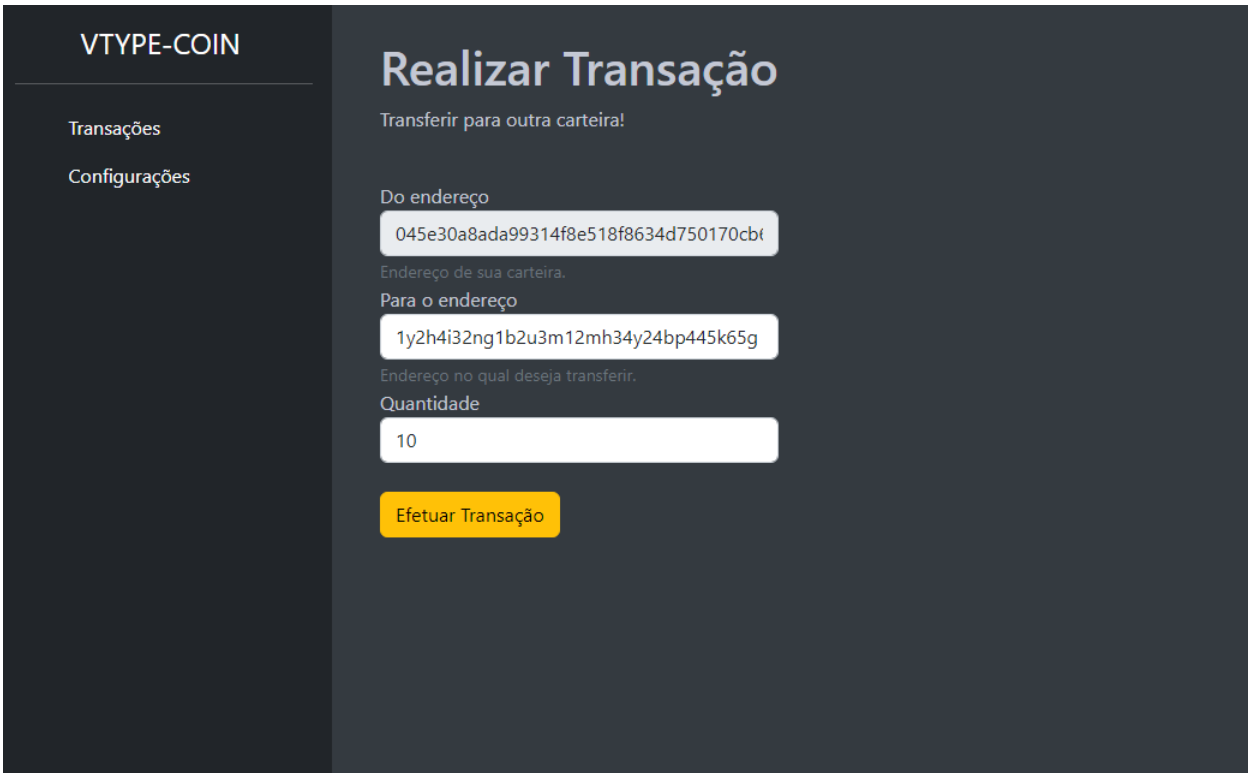
Figura 5.1 – Tela Inicial da Aplicação



Fonte: Do Autor

Ao clicar na opção de "Transações", o usuário é direcionado para a tela de transações, onde pode realizar transferências para outra carteira virtual. O usuário pode digitar o endereço da carteira de destino, selecionar a quantidade e efetuar a transação, tal como na Figura 5.2

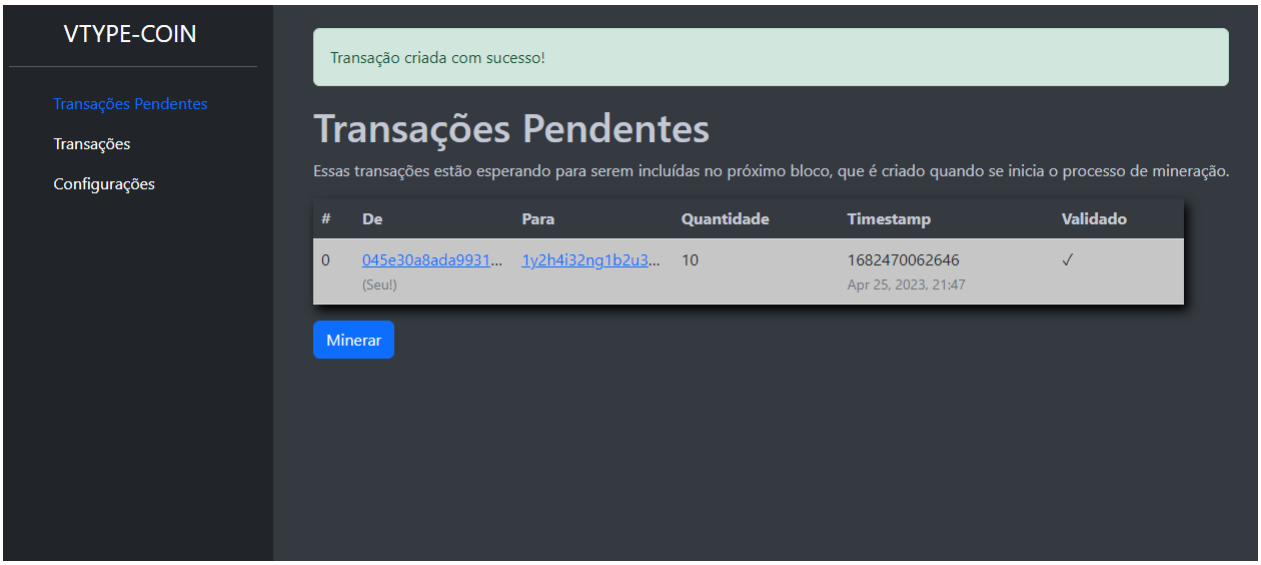
Figura 5.2 – Tela de Transações



Fonte: Do Autor

Ao clicar no botão de efetuar transação, a opção Transações Pendentes aparece no menu lateral e o usuário é direcionado para a tela de transações pendentes, onde ele consegue ver os dados da transação, além de fazer mineração, tal como na Figura 5.3

Figura 5.3 – Tela de Transações Pendentes



Fonte: Do Autor

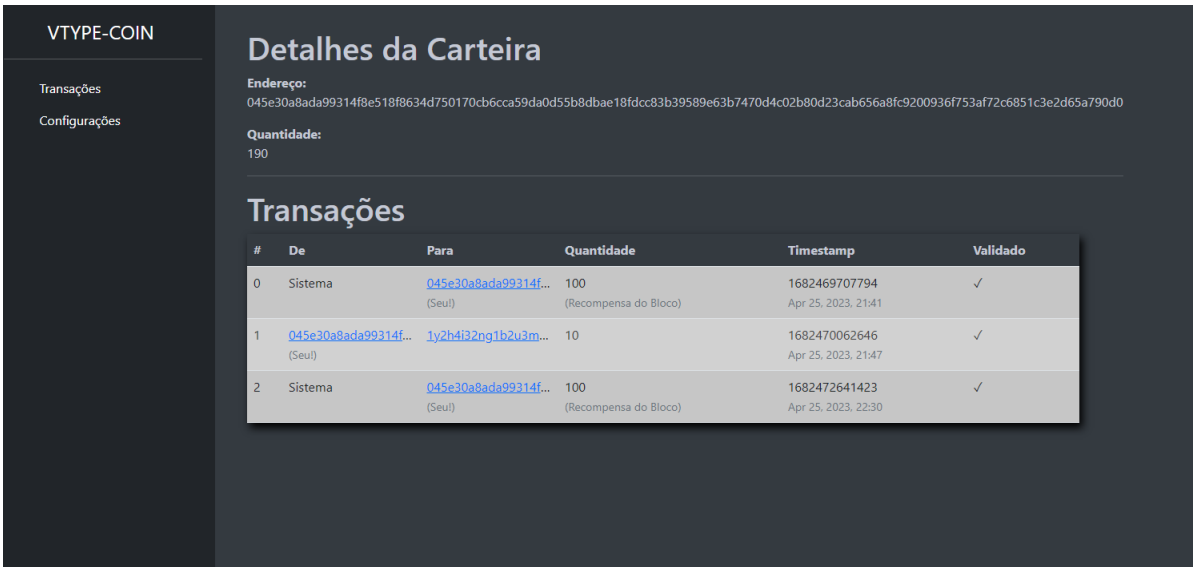
Ao realizar a mineração, o usuário é direcionado para a tela inicial novamente, de acordo com a Figura 5.4. Além disso, percebe-se que um novo bloco foi criado, e dentro dele há duas transações, a primeira é a transação feita, e a segunda, é a própria *blockchain* recompensando, com uma quantidade de 100 moedas, para quem fez a mineração.

Figura 5.4 – Tela Inicial pós Mineração

Fonte: Do Autor

Ao clicar no *link* do endereço da carteira dentro da tabela de transações do bloco, a interface é direcionada para a tela de detalhes da carteira, onde apresenta o endereço público da carteira, a quantidade de moedas que ela possui e as transações ligadas a ela, tal como mostra a Figura 5.5.

Figura 5.5 – Tela de Detalhes da Carteira



Fonte: Do Autor

Por fim tem-se a tela de configurações da *blockchain*, onde pode-se alterar a dificuldade do algoritmo de mineração e também a recompensa para os mineradores, tal como é mostrado na Figura 5.6.

Figura 5.6 – Tela de Configuração da Blockchain

Fonte: Do Autor

6 CONCLUSÃO

A *blockchain* é uma tecnologia que tem ganhado importância nos dias atuais, principalmente por sua capacidade de fornecer um registro seguro e confiável. Isso significa que, diferentemente de outras formas de armazenamento de dados, a *blockchain* permite armazenamentos seguros e transparentes, sejam eles para transações ou para outras finalidades. Essas características são importante em um mundo onde a segurança digital é uma preocupação crescente.

A *blockchain* tem sido usada para criar uma variedade de aplicativos que são considerados mais seguros e confiáveis do que as soluções convencionais. Portanto, uma implementação simples e objetiva de uma biblioteca capaz de gerenciar uma *blockchain* se faz útil para disseminar o conhecimento sobre o assunto e incentivar o uso dessa nova tecnologia que ano após ano avança e conquista seu devido espaço.

Fazer uma biblioteca para manipulação de uma *blockchain* foi uma tarefa desafiadora que envolveu diversos aspectos técnicos e principalmente conceituais. A seguir, são destacados alguns dos principais desafios enfrentados no processo de desenvolvimento da biblioteca.

- O Entendimento do conceito de *blockchain*: antes de começar a implementar uma biblioteca para manipulação de uma *blockchain*, foi necessário compreender em profundidade os conceitos envolvidos. Isso inclui entender como funciona a criptografia de chave pública, como é feita a validação de transações e blocos, e como os nós da rede se comunicam entre si para manter a integridade da *blockchain*.

- Implementação do algoritmo de consenso: uma das principais características de uma *blockchain* é o seu algoritmo de consenso, que é responsável por garantir que a rede inteira concorda sobre o estado atual da *blockchain*. Existem diversos algoritmos de consenso diferentes, cada um com suas vantagens e desvantagens. Implementar um algoritmo de consenso requer conhecimento avançado de criptografia e programação distribuída, por isso foi escolhido um algorítmo mais simples para esse caso, e com isso pôde-se obter um bom resultado na implementação.

- Gerenciamento de transações: como a *blockchain* é uma estrutura de dados que armazena transações, é importante garantir que apenas transações válidas sejam adicionadas à *blockchain*. Isso envolve validar as assinaturas digitais das transações, verificar se o remetente possui saldo suficiente para realizar a transação e evitar gastos duplos. Além disso, foi ne-

cessário implementar um mecanismo para lidar com transações pendentes e garantir que elas sejam adicionadas à *blockchain* no momento apropriado.

- Armazenamento e gerenciamento de dados: uma *blockchain* é uma estrutura de dados distribuída que é mantida por uma rede de nós. Isso significa que cada nó precisa armazenar uma cópia da *blockchain* inteira, por isso na implementação da bilbioteca, foi definido que a distribuição da *blockchain* ficaria a cargo do utilizador.

- Assinatura digital: por se tratar de uma das funcionalidades que garante a segurança das transações na *blockchain*, se faz necessário sua implementação, mas para isso, é necessário ter o conhecimento para aplicar corretamente;

- Segurança: como as *blockchains* podem ser usadas para armazenar e transferir valores, a segurança foi uma preocupação fundamental. Com isso, foi necessário implementar medidas para proteger a rede, e a medida escolhida foi a transação assinada digitalmente.

- Encontrar bons conteúdos: por ser um assunto atual, encontrar materiais gratuitos e que auxiliam no entendimento dos conceitos que a *blockchain* envolve foi muito difícil. Porém, mesmo com essa dificuldade, há materiais excelentes em inglês e alguns repositórios no github contém implementações que servem como base.

Para trabalhos futuros, é interessante distribuir a biblioteca em vez de pelo github, diretamente pelo NPM (NPM, 2023), que é o gerenciador de pacotes do Node (Node Package Manager), que vem junto com ele. Ademais, realizar algumas melhorias no código da *blockchain*, como o algoritmo de consenso, tornando-o mais complexo de ser resolvido, o que resulta em um ganho de segurança para a *blockchain*. Além disso, para facilitar ainda mais o uso da biblioteca pelo utilizador, pode-se desenvolver um mecanismo nativo capaz de fazer a *blockchain* operar de forma distribuída de acordo com suas necessidades.

Ao apresentar o desenvolvimento do projeto, se faz importante correlacioná-lo com disciplinas que foram estudadas ao longo do curso, disciplinas estas que ajudaram no desenvolver do projeto e contribuíram para o bom êxito da biblioteca. Dito isso, os conhecimentos obtidos em estrutura de dados foram cruciais para o entendimento de como a *blockchain* se conecta, sendo semelhante a uma lista encadeada. Ademais, o conhecimento relacionado a sistemas distribuídos deu uma percepção de como é a comunicação entre cada nó da *blockchain*, sendo feita com *sockets*. Por fim, a disciplina de práticas de programação orientada a objetos teve sua contribuição na organização de todo o código fonte do projeto, pois os conceitos aprendidos puderam servir de base para uma implementação limpa e organizada.

Encerrando o presente trabalho, é válido pontuar que o esforço para se aprender sobre algum assunto, é recompensado por um conhecimento que permanece em nossa memória. Conhecimento este que pode ser útil ou não, mas que com a nossa capacidade cognitiva, se torna uma ferramenta aplicável em qualquer área que possamos imaginar. Dito isso, toda dificuldade e esforço dedicado ao longo do curso, ajudou o autor deste trabalho a não só obter conhecimento e habilidades novas, mas principalmente a ir mais longe, tendo vontade de aprender novas tecnologias e disseminar o conhecimento adquirido, contribuindo assim em uma pequena parcela para a evolução da computação.

REFERÊNCIAS

101 BLOCKCHAINS. 2018. (Acessado em 30/06/2023). Disponível em: <https://101blockchains.com/pt/historia-da-tecnologia-blockchain/>.

ANGULAR. 2023. (Acessado em 25/04/2023). Disponível em: <https://angular.io/>.

GITHUB. 2023. (Acessado em 24/04/2023). Disponível em: <https://github.com/>.

HABER, S., Stornetta, W. S. How to Time-Stamp a Digital Document. 1991. (Acessado em 24/04/2023). Disponível em: <http://www.staroceans.org/e-book/Haber_Stornetta.pdf>.

NAKAMOTO, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. (Acessado em 24/04/2023). Disponível em: <https://bitcoin.org/bitcoin.pdf>.

NPM. 2023. (Acessado em 25/04/2023). Disponível em: <https://www.npmjs.com/>.

YOUTUBE. 2023. (Acessado em 24/04/2023). Disponível em: <https://www.youtube.com/>.