DonationChain: Aplicação descentralizada para doação de criptomoedas

Vinicius B. Meneses, Carlos Eduardo Pagani

Instituto Federal de São Paulo – Câmpus Hortolândia

vinicius.meneses04@gmail.com, pagani@ifsp.edu.br

Abstract. *This paper aims to help charities struggling with financial challanges caused by the outcomes of the COVID-19 pandemic, during a time that the digital currency market stands out for its growth. A decentralized application was developed to promote and allow cryptocurrency donations straight to those various nonprofits. The web platform was implemented based on the blockchain technology along with resources from the Binance Smart Chain network and using open-source tools, ensuring high availability and reliability.*

Resumo. *Este trabalho visa ajudar instituições de caridade que enfrentam dificuldades financeiras devido às consequências da pandemia de COVID-19, durante um período em que o mercado de moedas digitais se destaca por apresentar grande crescimento. Foi desenvolvida uma aplicação descentralizada com o intuito de incentivar e possibilitar a doação de criptomoedas diretamente para as entidades sem fins lucrativos que apoiam as mais diversas causas. A plataforma, projetada para web, foi implementada baseando-se na tecnologia blockchain em conjunto com os recursos da rede Binance Smart Chain e utilizando ferramentas de código aberto, garantindo alta disponibilidade e confiabilidade na sua utilização.*

1. Introdução

Devido à pandemia de COVID-19[1] que se iniciou no fim de 2019, diversas ONGs (Organizações não Governamentais) e associações solidárias no Brasil começaram a passar por dificuldades financeiras. A principal causa desse problema é a diminuição no número de doações, pois como grande parte da população brasileira foi afetada economicamente por esse novo vírus e, assim, muitos pararam de realizar contribuições a instituições de caridade pela falta de renda.

O estudo Impacto da COVID-19 nas OSCs (Organizações da Sociedade Civil) brasileiras (Mobiliza, ReosPartners, 2020) apurou que 73% de 1760 ONGs entrevistadas foram impactadas de alguma forma pela pandemia, num cenário onde 89% das instituições estão oferecendo ajuda a pessoas afetadas por esse novo vírus. O impacto negativo mais relatado pelos entrevistados é a queda no volume de doações, sendo que 60% deles preveem que haja um aumento na demanda pelos serviços realizados por essas associações.

Por outro lado, as criptomoedas vêm ganhando cada vez mais popularidade nos últimos anos, aumentando o número de pessoas que se interessam, possuem e até mesmo utilizam as moedas digitais no dia a dia como forma de pagamento.

Apesar dos diversos problemas econômicos causados pela pandemia de COVID-19 no mundo e, principalmente, nas instituições de caridade, o mercado de moedas digitais não entrou em queda, muito pelo contrário, recebeu mais investidores que o normal durante todo esse período. Por meio do site especializado em monitoramento de criptomoedas

[1] https://www.paho.org/pt/news/11-3-2020-who-characterizes-covid-19-pandemic

CoinMarketCap, é possível notar que a capitalização de mercado das moedas digitais saltou de 192 bilhões de dólares no início de 2020 para mais de 1 trilhão de dólares em junho de 2021, um aumento de 690% (CoinMarketCap, 2021).

Diante do grande crescimento do mercado de criptomoedas mesmo no contexto da pandemia e a falta de recursos financeiros para as ONGs brasileiras manterem suas atividades, devido à queda no número de doações realizadas, o objetivo deste trabalho é desenvolver uma aplicação descentralizada denominada *DonationChain*, que irá funcionar sobre uma rede *blockchain* e armazenará todos os seus dados nela, a fim de incentivar as pessoas detentoras de criptomoedas a realizarem doações a ONGs conhecidas, ampliar a usabilidade das moedas digitais e possibilitar às instituições de caridade abrangerem um novo grupo de possíveis doadores. A aplicação permitirá que os usuários doem suas moedas que estão na sua carteira da rede diretamente para a instituição de caridade, sem que antes precise convertê-las em moeda fiduciária para efetivar a contribuição.

2. Referencial Teórico

A fim de possibilitar a implementação de uma aplicação *web* descentralizada que permita doar criptomoedas, é necessário que ela se comunique com uma rede que implemente a tecnologia *blockchain*. Essa tecnologia possui forte ligação com as funções *hash*, pois tais algoritmos são amplamente empregados na implementação da *blockchain*.

Ademais, é necessário que a rede utilizada também empregue o conceito de contrato inteligente, como a *Ethereum*, para possibilitar o desenvolvimento de uma aplicação *web* descentralizada, que possui várias diferenças em sua arquitetura e na modelagem de seus dados quando comparadas a maioria dos sistemas *web* existentes.

Portanto, nas próximas subseções será apresentado o referencial teórico utilizado para o desenvolvimento da aplicação proposta neste trabalho.

2.1 Funções *Hash*

As funções *hash* criptográficas são algoritmos unidirecionais responsáveis por receber uma mensagem de comprimento variável como entrada e convertê-la em uma saída de tamanho fixo geralmente chamada de resumo ou simplesmente *hash*. Por serem unidirecionais, é praticamente impossível descobrir a entrada utilizada somente por meio do resumo gerado, a única maneira de descobrir a mensagem original é por meio da tentativa e erro, processando todas as mensagens possíveis pela função *hash* utilizada para encontrar a saída que seja igual ao *hash* (Stallings, 2014).

As funções *hash* também são determinísticas, por isso, a mesma entrada sempre irá gerar o mesmo *hash*, e qualquer alteração na mensagem irá gerar uma saída totalmente diferente, não sendo possível correlacionar o novo *hash* com o antigo. Além disso, elas resolvem o princípio de integridade da criptografia, garantindo que determinada mensagem não foi alterada durante a transmissão entre o remetente e o destinatário (Souza, 2020).

Na Figura 10 do Anexo 1 são apresentados alguns exemplos de mensagens processadas por determinada função *hash* e a saída gerada para cada entrada.

Os algoritmos de funções *hash* mais conhecidos são o MD5, pertencente à família MD, SHA-1, SHA-2 e SHA-3, referentes à família SHA. Os algoritmos da família MD, eram conhecidos por gerarem um resumo de 128 bits (32 dígitos em formato hexadecimal) e foram muito utilizados para verificar a integridade de arquivos, mas encontraram-se diversos problemas de vulnerabilidade fazendo com que seu uso fosse descontinuado. Já a família SHA

(*Secure Hash Algorithm*) é composta de 4 algoritmos. O SHA-0 foi o primeiro algoritmo desenvolvido em 1993, que foi descontinuado rapidamente por falhas de segurança e substituído pelo SHA-1, projetado e publicado pela Agência de Segurança Nacional dos Estados Unidos em 1995. Após ser utilizado em diversos protocolos da Internet, como SSH, SSL e TSL, e em ferramenta de controles de versão de código, ele foi descontinuado em 2010 para dar lugar ao SHA-2 e posteriormente, em 2015, ao SHA-3 (Singhal, Dhameja, Panda, 2018).

O SHA-2 por si só é uma outra família de algoritmos também pertencentes à família SHA, sendo o SHA-256 o mais conhecido e utilizado na maioria das redes *blockchain*, ele produz um *hash* de 64 dígitos em formato hexadecimal.

2.2 *Blockchain*

Blockchain é uma tecnologia que permite criar uma base de dados permanente e compartilhável, onde as informações inseridas nela não podem ser alteradas e removidas, por esse motivo, sempre que algum usuário desejar incluir um novo dado na *blockchain*, é necessário que ele crie uma transação, que irá propor a adição da informação e será validada pelos outros usuários (Laurence, 2017).

Essa tecnologia não possui uma autoridade, como um banco central ou órgão governamental, que gerencia o fluxo das informações. Ela é formada por uma grande rede ponto-a-ponto de usuários independentes que garantem a integridade dos dados. Cada ponto conectado (*full-node*) possui uma cópia da base de dados e tem o poder de validar as transações submetidas pelos outros nós, a fim de aprovar ou não as inserções de dados na *blockchain*, além de também ser responsável por compartilhar o banco de dados com os demais participantes para manter a rede atualizada (Laurence, 2017).

A tecnologia *blockchain* se baseia em alguns conceitos importantes para que seu propósito seja cumprido, sendo eles (Singhal, Dhameja, Panda, 2018):

- *Blockchain* é um sistema ponto-a-ponto para transacionar dados sem uma entidade externa confiável.
- É um registro de transações compartilhado e descentralizado. Esse banco de dados é replicado em um grande número de nós da rede.
- Esse registro de dados só suporta adições, os dados não podem ser alterados ou removidos. Dessa forma, cada nova inserção é uma inserção permanente, que será refletida em todas as cópias da base de dados hospedados em diferentes nós.
- Não há a necessidade de uma autoridade central para validar transações, proteger a rede ou servir como intermediário.
- A tecnologia *blockchain* funciona em uma outra camada acima da internet e pode coexistir com várias outras tecnologias.

Para a *blockchain* se tornar um registro de dados imutável, várias transações validadas durante um determinado período, responsáveis por inserir ou sobrescrever dados, são agrupadas em uma estrutura denominada bloco (Figura 11 do Anexo 1).

Cada bloco possui um *hash*, identificador gerado por uma função *hash* criptográfica a partir das transações contidas no bloco, e é capaz de reunir uma determinada quantidade máxima de transações, sendo de responsabilidade da *blockchain* em questão definir o seu tamanho e o intervalo de tempo que necessário para que um novo bloco possa ser criado. Através do *hash* gerado, eles acabam sendo encadeados “matematicamente”, tornando-se

possível determinar facilmente qual é o bloco antecedente e como a base de dados era antes do bloco atual ser criado, dando sentido ao termo *blockchain* (Nakamoto, 2008). Na Figura 12 do Anexo 1 é ilustrado um exemplo de três blocos sendo conectados por meio de seus *hashes* formando uma cadeia.

2.2.1 Consenso

A tecnologia *blockchain* não necessita de uma entidade externa para validar as transações ou assegurar que as suas regras estão sendo cumpridas, pois ela é capaz de criar uma rede segura, que se "autocorrige", caso haja tentativas de burlar o sistema, sem a necessidade de um terceiro, por meio do mecanismo de consenso utilizado. No contexto da *blockchain*, consenso é o processo de encontrar um mesmo acordo entre os nós da rede que estão validando as transações, dessa forma, somente se os nós concordem que um grupo de transações são válidas, entrando em um consenso, elas serão registradas na base de dados (Laurence, 2017).

As redes *blockchain* precisam ser ágeis na maioria dos casos, de forma que as transações sejam validadas e compartilhadas com confiabilidade e no menor espaço de tempo possível, por isso, os algoritmos de consenso sempre atuam sobre um determinado grupo de transações para validá-las, e não individualmente. Para que isso seja possível, somente um nó da rede de cada vez pode propor um novo bloco, que ele acredita conter transações válidas, e todos os outros nós devem validá-lo, adicionando as transações nas suas respectivas bases de dados se concordarem que elas são válidas. Entretanto, a fim de manter a rede organizada, a escolha do nó que irá propor o novo bloco é feita por meio do algoritmo de consenso implementado na *blockchain* (Laurence, 2017). Atualmente, existem dois principais mecanismos de consenso, que são utilizados nas maiores redes, sendo eles *Proof of Work* (PoW) e *Proof of Stake* (PoS).

O algoritmo PoW é baseado na ideia de que um processamento computacional deve ser feito utilizando os dados de determinado grupo de transações, com o objetivo de encontrar uma resposta que esteja de acordo com os requisitos de validação da *blockchain* em questão, provando que o nó gastou tempo e poder computacional para validar o bloco, antes que ele possa ser proposto para toda a rede. Por outro lado, após um nó encontrar a resposta, ela pode ser facilmente validada pelos outros nós da rede utilizando os mesmos métodos e variáveis, somente comparando se a resposta produzida é igual a resposta encontrada pelo criador do bloco (Abijaude, Greve, Sobreira, 2021).

Em *blockchains* públicas que utilizam o PoW, é preciso que haja uma recompensa para incentivar os nós a investirem seu poder computacional com o objetivo de resolver esse "quebra-cabeça" e validar um grupo de transações, esse processo é geralmente chamado de mineração, e previne que algum participante malicioso tente propor um bloco inválido, pois ele irá gastar tempo e recursos, mas não irá receber a recompensa no final (Singhal, Dhameia, Panda, 2018).

A cada resposta válida encontrada para uma quantidade de transações, um novo bloco é criado, mas para que se torne possível controlar a velocidade de blocos produzidos durante certo tempo, é utilizado o conceito de dificuldade, que permite aumentar ou diminuir o tempo e os recursos necessários para encontrar o resultado. Caso haja muitos nós conectados à rede tentando propor blocos para receber as recompensas, a dificuldade pode ser ajustada a fim de exigir mais tempo para encontrar o resultado de acordo com as regras da *blockchain*, diminuindo a velocidade de criação de blocos. Por outro lado, é possível diminuir a dificuldade quando houver um pequeno número de pontos na rede, aumentando a quantidade de blocos produzidos durante um intervalo de tempo (Singhal, Dhameia, Panda, 2018).

Nas *blockchains* que utilizam o algoritmo PoW, todo o processo de utilizar poder computacional com a finalidade de encontrar uma resposta, conforme as exigências da rede, para incluir um novo bloco é feito por meio de tentativa e erro utilizando funções *hash*. Os dados do bloco a ser criado são submetidos, em conjunto com uma variável denominada "*nonce*", a uma função *hash* e verifica-se se o *hash* produzido está de acordo com as regras da *blockchain*. Se o resumo produzido não for válido, novas tentativas devem ser feitas mantendo os dados do bloco, mas alterando o valor do *nonce*, que é simplesmente um valor aleatório, até que encontre o *hash* válido (Singhal, Dhameia, Panda, 2018).

Como as funções *hash* são determinísticas, não seria possível utilizar somente os dados do bloco para produzir os *hashes*, pois eles sempre estariam iguais. Por esse motivo é necessário o uso da variável *nonce*, visto que qualquer mudança nos dados de entrada, mesmo que mínima, irá gerar um *hash* totalmente diferente (Greve *et al*, 2018).

Já no algoritmo PoS não há o conceito de mineração, que concede recompensas ao nó que validou um bloco, somente existem nós validadores, responsáveis por "forjar" novos blocos em troca de taxas cobradas dos nós que desejarem propor transações. Para um participante se tornar validador e ter a chance de validar um bloco, ele precisa manter certa quantidade de moedas bloqueadas na rede (em *stake*) e a *blockchain*, em determinados intervalos de tempo, irá atribuir aleatoriamente o direito de produzir o próximo bloco a algum dos validadores, recebendo as taxas de transação cobradas em troca (Greve *et al*, 2018). A probabilidade de um validador ser escolhido pela rede é proporcional à quantidade de moedas bloqueadas, quanto mais moedas em *stake*, maior a chance de ele ser selecionado dentre os demais participantes (Singhal, Dhameja, Panda, 2018).

Esse algoritmo provê uma proteção melhor contra nós maliciosos, visto que os validadores que tentarem manipular a rede poderão perder todas as suas moedas em *stake*, e incentiva ainda mais a descentralização da rede, pois irão existir mais validadores, já que não é necessário um hardware poderoso para realizar o processamento computacional como no PoW.

2.3 *Ethereum*

A *blockchain* essencialmente se resume em realizar transições de estado. As alterações de estado realizadas na *blockchain* podem ser visualizadas por qualquer pessoa, mas qualquer alteração de estado solicitada por um participante da rede só pode ser efetivada após passar pelo processo de verificação e consenso. Por exemplo, em uma rede de determinada moeda digital que utiliza essa tecnologia, o estado global da *blockchain* é alterado sempre que um usuário envia determinada quantia de moedas para outro usuário (Peyrott, 2017). Na Figura 13 do Anexo 1 é exemplificado a mudança de estado quando o usuário A enviar 50 moedas para o usuário B.

Visto que as aplicações comuns para computadores sempre lidam, de alguma forma, com transições de estado e a tecnologia *blockchain*, sendo genérica, possibilita encontrar o consenso independentemente do processamento computacional realizado, a rede *Ethereum* surge com o propósito de permitir que programas sejam desenvolvidos e publicados na rede, possibilitando realização de alterações no estado da rede de forma genérica por meio de transações.

A plataforma *Ethereum* viabiliza o suporte a aplicações criando uma camada de abstração para manipular o estado da *blockchain* que podem ser utilizadas no desenvolvimento de qualquer tipo de algoritmo. Dessa forma, as pessoas podem desenvolver

programas que possuem as mais variadas funcionalidades com a ajuda da abstração disponibilizada pela plataforma. Ela simplifica as dificuldades e particularidades de uma rede descentralizada e permite que o próprio desenvolvedor do programa projete suas funções que irão alterar o estado da rede (Singhal, Dhameja, Panda, 2018).

Essa *blockchain* atualmente utiliza o algoritmo de consenso PoW, em que os nós competem entre si para criar blocos a partir de transações validadas. O participante da rede que utilizar seu poder computacional e for o primeiro a encontrar a resposta desejada pelo algoritmo, receberá uma certa quantia de moedas digitais como recompensa e terá o direito de compartilhar o novo bloco com o restante da rede (Ethereum, 2021).

2.3.1 *Ethereum Virtual Machine*

A fim de cumprir os objetivos propostos, a principal inovação da *Ethereum* foi a *Ethereum Virtual Machine* (EVM), que é uma máquina virtual utilizada para interpretar trechos de código escritos com algumas linguagens de programação. Da mesma forma que a *Java Virtual Machine* (JVM) é necessária para executar códigos escritos em Java, a EVM é requerida para rodar *scripts* escritos em linguagens de programação suportadas pela plataforma, como a linguagem *Solidity*. (Peyrott, 2017).

Na *Ethereum*, cada programa codificado e publicado poderá ser executado por qualquer nó da rede, pois cada ponto possui a cópia completa da *blockchain* e uma instância da EVM, capaz de executar o código dos programas quando ele desejar validar uma nova transação. A rede executa a aplicação especificada na transação, que irá realizar determinado processamento e modificará o estado da *blockchain* (Singhal, Dhameja, Panda, 2018).

2.3.2 *Ether*

O *Ether* é a moeda digital nativa da rede *Ethereum,* que pode ser armazenada pelos participantes, transferida ou recebida como recompensa por validar transações e criar blocos.

Os códigos publicados na rede *Ethereum* podem implementar seja qual for funcionalidade e realizar qualquer tipo de processamento, e, por esse motivo, eles são suscetíveis ao Problema da Parada, que diz respeito a impossibilidade de saber se, dado uma entrada, o programa irá finalizar sua execução em algum momento ou ficará executando infinitamente. Dado esse problema, o *Ether* também surge com o propósito limitar o tempo máximo de execução que determinada funcionalidade terá para ser executada completamente na EVM, evitando ataques maliciosos à rede e funcionalidades codificadas de forma errada, a qual poderiam rodar infinitamente (Peyrott, 2017).

A cada vez que uma função for chamada, o usuário que solicitou a execução deve definir o máximo de *Ether* que pode ser gasto na transação (*gas*), dessa forma, o *Ether* é consumido enquanto o código é executado e a EVM poderá encerrar o programa a qualquer momento caso falte *gas* (Peyrott, 2017).

2.3.3 Contrato Inteligente

Contrato inteligente, ou *smart contract,* é o nome dado a qualquer programa desenvolvido para ser executado numa *blockchain*, eles são o elemento principal da rede *Ethereum* (Peyrott, 2017). Os usuários podem interagir com um contrato inteligente enviando transações que executam determinada função dele. Da mesma forma que um contrato comum, os contratos inteligentes também podem definir e assegurar que as regras estão sendo cumpridas, porém esse processo é feito automaticamente através de linhas de código. Os programas, uma vez

publicados, não podem ser excluídos da rede e as interações com eles são irreversíveis. (Ethereum, 2021)

2.4 *Binance Smart Chain*

A *Binance Smart Chain*, desenvolvida pela *Binance*, a maior plataforma de negociação de criptomoedas do mundo, é uma *blockchain* compatível com a EVM criada para suportar contratos inteligentes.

Essa rede é focada em tornar as aplicações descentralizadas acessíveis a todos, provendo um maior limite máximo de *gas* que o usuário pode definir e menor tempo de geração de blocos. Sua arquitetura é baseada fortemente na rede *Ethereum* e na EVM, possibilitando que projetos e ferramentas desenvolvidas para a *Ethereum* também funcionem nativamente nessa rede (Bison Trails, 2021).

Assim como a *Ethereum*, a *Binance Smart Chain* também possui a *Binance Coin* como criptomoeda nativa. Ela tem praticamente as mesmas utilidades do *Ether*, ou seja, pode ser transferida, armazenada ou utilizada para pagamento de taxas. Além disso, cerca de 90% dos funcionários da *exchange Binance* recebem parte do seu salário nessa moeda.

O consenso entre os validadores sobre o estado da *blockchain* na *Binance Smart Chain* é atingido por meio do algoritmo *Proof of Staked Authority* (PoSA), uma combinação dos mecanismos PoS e *Proof of Authority* (PoA) (Binance, 2021). Os validadores da rede realizam o *stake* de certa quantia de *Binance Coin* como garantia e, quando é proposto um novo bloco válido, o participante recebe todas as taxas das transações contidas naquele bloco. Por também utilizar o algoritmo PoA, a *blockchain* foi desenvolvida para suportar somente até 21 validadores, e os nós que irão validar transações são definidos por meio da sua autoridade na rede, ou seja, quanto maior a quantidade de *Binance Coin* que o nó adicionou em *stake*, maior sua chance de se tornar um validador (Bison Trails, 2021).

A Tabela 1 realiza uma comparação entre a rede *Binance Smart Chain* e a *Ethereum*, mostrando as principais diferenças entre si e suas características em comum.

Tabela 1. Comparação entre as redes *Binance Smart Chain* e *Ethereum*

Redes	Binance Smart Chain	Ethereum
Endereços ativos (aprox.)	2.000.000	800.000
Transações diárias (aprox.)	4.000.000	1.250.000
Taxa de transação (aprox.)	$ 0,05	$ 5,94
Tempo para um novo bloco ser criado	3 segundos	13 segundos
Algoritmo de consenso	PoSA	PoW
Criptomoeda nativa	Binance Coin	Ether
Linguagem utilizada	Solidity	Solidity
IDE utilizada	Remix	Remix

Visto que essa rede é focada em aplicações descentralizadas, ela possui dois ambientes: *mainnet* e *testnet*. O *mainnet* é o ambiente oficial da *Binance Smart Chain*, que conta com todas as suas características originais, ou seja, os validadores são entidades confiáveis escolhidas utilizando o algoritmo de consenso implantado e todas as criptomoedas possuem valor financeiro. Já o *testnet* é o ambiente de testes da rede mantido pela comunidade, ele é disponibilizado para que as pessoas publiquem contratos inteligentes a fim de testá-los antes de serem lançados oficialmente. Os validadores do ambiente de testes é o próprio time de desenvolvimento da *Binance* e as moedas digitais que circulam pela rede não possuem qualquer valor (Binance, 2021).

2.5 Aplicação descentralizada

Aplicação descentralizada é uma aplicação que funciona sobre uma rede descentralizada e constitui-se de contratos inteligentes e interfaces de usuário, para facilitar o uso dos contratos (Ethereum, 2021).

Numa aplicação comum, todos os dados do sistema são armazenados em servidores centralizados e gerenciados pelo dono da aplicação, porém, em aplicações descentralizadas, todas as informações são salvas publicamente numa *blockchain* descentralizada (Ethereum, 2021).

De forma geral, uma aplicação deve possuir as seguintes características para ser considerada descentralizada (Ethereum, 2021):

- Operam numa plataforma pública e descentralizada onde nenhuma pessoa ou grupo tem controle.
- Elas desempenham as mesmas funcionalidades independentemente do contexto em que são executadas.
- Podem realizar qualquer ação com os recursos necessários.
- Deve ser possível executá-las utilizando o ambiente virtual da EVM, pois se houver algum problema no código do contrato inteligente, ele não afetará o funcionamento da rede.

As aplicações, para se tornarem descentralizadas, não precisam necessariamente funcionar sobre a rede *Ethereum*. Elas podem ser implementadas em qualquer rede descentralizada e que suporte contratos inteligentes.

2.6 *Solidity*

Solidity é uma linguagem de programação de alto-nível que possibilita o desenvolvimento de contratos inteligentes e a compilação deles para linguagem de máquina para serem interpretados pela EVM. Além da *Solidity*, também existem outras linguagens que podem ser utilizadas na criação de contratos, como *Serpent* e LLL (*Lisp Like Language*), porém a comunidade foi aos poucos deixando de usá-las e focando somente na *Solidity* (Dannen, 2017).

Ela é uma linguagem tipada e orientada a objetos, que suporta herança e bibliotecas, e, por meio dela, torna-se possível a implementação de contratos inteligentes para serem usados em votações, leilões, financiamentos coletivos etc. (Ethereum, 2021).

Na Figura 14 do Anexo 2 é exemplificado um contrato inteligente desenvolvido com a linguagem Solidity, que armazena e manipula um contador.

2.7 *Remix*

É um ambiente de desenvolvimento de código aberto usado durante todo o processo de desenvolvimento de um contrato inteligente, desde o desenvolvimento, utilizando a linguagem *Solidity*, até a publicação na rede. O *Remix* é uma aplicação *web* e *desktop* com interfaces simples e intuitivas que acelera o desenvolvimento de contratos inteligentes e serve como um ambiente para ensinar como programar utilizando a linguagem *Solidity* (Remix, 2021). A Figura 15 do Anexo 2 mostra a interface inicial do Remix ao ser acessado pela primeira vez.

2.8 Padrões *Web*

HTML (*HyperText Markup Language*) é uma linguagem de marcação utilizada no desenvolvimento *web* para definir a estrutura básica de uma determinada página. Ela permite que sejam declarados uma série de elementos, como parágrafos, imagens, títulos etc., que juntos formarão a estrutura da página *web* a ser interpretada pelo navegador (MDN Web Docs, 2021).

Para estilizar os elementos declarados com o HTML, é utilizada a linguagem de estilo CSS (*Cascading Style Sheets*). Por meio dessa linguagem também é possível definir como os elementos devem ser apresentados na tela do dispositivo e a posição de cada um deles (MDN Web Docs, 2021).

O *JavaScript* é a principal linguagem de programação para desenvolvimento *web*, ela possibilita controlar os elementos da página HTML, adicionando comportamentos e reações ao serem interagidos pelo usuário (MDN Web Docs, 2021).

Entretanto, com o passar dos anos surgiram bibliotecas para facilitar o controle dos elementos visuais, tornando o desenvolvimento de interfaces gráficas mais rápido. O *React,* que foi desenvolvido pelo Facebook e lançado oficialmente em 2013, é um exemplo dessas bibliotecas. Ele visa facilitar o desenvolvimento de interfaces gráficas auxiliando no controle dos elementos da página dividindo-os em "componentes" reutilizáveis e com estado próprio (Copes, 2019).

Por fim, *Mantine* é uma outra biblioteca de código aberto escrita em *Javascript*, que contém diversos componentes visuais personalizáveis prontos para serem utilizados em conjunto com o *React,* auxiliando no desenvolvimento de interfaces gráficas com boa usabilidade (Mantine, 2021).

2.9 *Web3.js*

Web3.js é uma biblioteca *JavaScript* a qual possibilita os programadores a interagirem com *blockchains* que implementam a EVM, como a *Ethereum* e a *Binance Smart Chain*. Ela abstrai o acesso à rede por meio de funções que podem ser facilmente utilizadas em navegadores *web*, auxiliando na construção de aplicações descentralizadas (Beyer, 2019).

2.10 *Metamask*

É uma carteira de criptomoedas instalada no próprio navegador *web* que suporta qualquer rede *blockchain* baseada na EVM. Uma vez instalada e configurada, o usuário pode armazenar moedas, realizar transferências e interagir com aplicações descentralizadas. Para possibilitar a interação com as aplicações, a *Metamask*[2] expõe uma "ponte" que é utilizada pela biblioteca Web3.js, a fim de conectar-se com a *blockchain*, obter dados públicos armazenados na rede e solicitar que o usuário aprove transações (Beyer, 2019).

2.11 Modelo de Desenvolvimento

Modelo de processo é um conjunto de regras e padrões definidos para guiar o desenvolvimento de um *software* e facilitar a construção do projeto. Além disso, é um plano para ajudar a entregar uma aplicação de alta qualidade e dentro do prazo estabelecido (Pressman, 2011).

[2] https://metamask.io

Um dos modelos de processos existentes é o incremental, o principal objetivo desse modelo é traçar sequências lineares de entrega, em que cada sequência é gerada um novo incremento de *software*, visando entregar ao usuário ou cliente um produto funcional do *software* a cada incremento. De forma geral, esse modelo é utilizado no desenvolvimento de uma aplicação quando os requisitos iniciais do *software* estão bem definidos e é preciso fornecer, o mais rápido possível ao usuário, determinado conjunto de funcionalidades para obter *feedbacks* a serem considerados na produção do próximo incremento (Pressman, 2011).

2.12 *Unified Modeling Language*

A *Unified Modeling Language* (UML) é uma linguagem-padrão para especificar, descrever, modelar e documentar *softwares*. Ela é composta de 13 diagramas no total, sendo os diagramas de classe, atividade, caso de uso e sequência os mais conhecidos (Fowler, 2003).

3. Trabalhos correlatos

Nesta seção serão demonstrados alguns trabalhos correlatos, que são projetos ou aplicações com ideias parecidas em relação a proposta deste trabalho, utilizados para auxiliar no desenvolvimento de ideias aplicadas no projeto do artigo.

3.1 *Giveth.io*

Giveth.io é uma aplicação web descentralizada de código aberto que possibilita pessoas detentoras de criptomoedas a realizarem doações a projetos sociais publicados por qualquer usuário da plataforma. Esse sistema utiliza a rede *Ethereum* para receber e armazenar o histórico de doações, dessa forma, os doadores devem possuir *Ether* ou alguma moeda suportada pela rede para poder realizar a contribuição (Giveth, 2021). A Figura 6 mostra a página principal de doação para um projeto social, na qual é possível contribuir utilizando criptomoedas ou cartão de crédito.

3.2 *The Giving Block*

Fundada em 2018, *The Giving Block* é uma plataforma web especializada em conectar doadores a instituições de caridade por meio das doações de moedas digitais. Essa solução permite que as pessoas doem diversas criptomoedas de redes diferentes para as organizações sem fins lucrativos cadastradas. As criptomoedas recebidas pelas ONGs são automaticamente convertidas em moeda fiduciária e os doadores recebem recibos para comprovar a contribuição realizada (The Giving Block, 2018).

4. Metodologia

Primeiramente, a arquitetura da aplicação foi definida baseando-se principalmente no conceito de aplicação descentralizada, visto que há uma grande diferença na forma entre as aplicações normais e as descentralizadas, principalmente porque ela interage com um contrato inteligente.

Em seguida, os requisitos foram levantados baseando-se na rede *Binance Smart Chain* e em aplicações semelhantes que possibilitam doação de criptomoedas para projetos sociais ou instituições de caridade. Ademais, foi documentada a modelagem dos dados necessários para o cumprimento dos requisitos da aplicação, bem como as relações que os dados possuem entre si e a forma que eles serão armazenados no estado da rede.

Por meio do modelo incremental, a aplicação foi elaborada em incrementos, onde a cada ciclo os requisitos foram aprimorados e as funcionalidades propostas foram desenvolvidas e validadas.

Após essa fase inicial, o contrato inteligente responsável por controlar a aplicação web foi implementado utilizando a linguagem de programação *Solidity* e publicado no ambiente de testes da rede *Binance Smart Chain*. Para cada funcionalidade do aplicativo criou-se uma função contrato inteligente, portanto, existem funções para permitir a doação de criptomoedas e incluir, visualizar, alterar e remover ONGs que podem receber contribuições, além de armazenar informações sobre todas as doações já realizadas.

A plataforma web, utilizada pelos usuários e responsável por manipular contrato criado, foi desenvolvida utilizando as principais tecnologias para aplicações web, sendo elas HTML, CSS, *JavaScript* e a biblioteca *React*. Com o intuito de integrar a aplicação com o contrato inteligente, usou-se a carteira *Metamask*, para armazenar as criptomoedas, em conjunto com a biblioteca *Web3.js*.

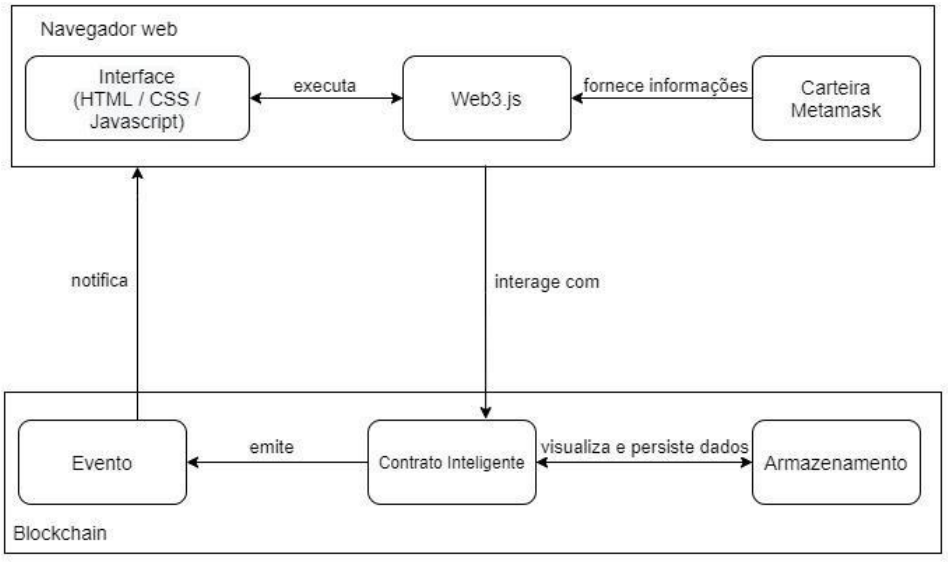
Ao final, o código, tanto da aplicação web quanto do contrato inteligente, foi liberado na internet para torná-lo um *software* de código aberto[3], e a aplicação web foi disponibilizada na internet por meio de um domínio, podendo ser acessada por qualquer dispositivo.

5. Desenvolvimento do trabalho

Esta seção apresenta a arquitetura e modelagem utilizada, os requisitos levantados e o processo de desenvolvimento do trabalho.

5.1 Arquitetura

Como descrito neste artigo, a aplicação será descentralizada e se comunicará diretamente com a rede *Binance Smart Chain* por meio de um contrato inteligente, portanto, a arquitetura dela se difere das arquiteturas na maioria das vezes utilizadas no desenvolvimento de sistemas *web*. Na Figura 1 é possível observar uma representação da arquitetura utilizada no desenvolvimento da aplicação.



[3] https://github.com/viniciusmeneses/donation-chain

Figura 1. Representação da arquitetura implementada nesta aplicação

A arquitetura é dividida em dois contextos, o primeiro sendo o navegador web do usuário e o segundo a rede *blockchain* que contém o contrato inteligente e armazena os dados da aplicação.

Utilizando a interface *web* do sistema, o usuário pode interagir com o contrato inteligente para visualizar informações persistidas e executar funções que alteram ou incluem dados no estado da *blockchain*. Quando o contrato inteligente recebe uma "chamada", uma das suas funções é executada e o estado da rede é manipulado. Após sua execução, eventos são emitidos notificando a aplicação que o contrato conseguiu executar com sucesso determinada funcionalidade.

Toda a comunicação realizada entre a interface *web* e a *Binance Smart Chain* é feita por meio da biblioteca *Web3.js*, que permite enviar e visualizar transações, manipular contratos inteligentes e obter informações sobre determinada carteira.

Em redes baseadas na *Ethereum*, os nós fornecem uma interface pública de baixo nível utilizando o protocolo *JSON-RPC* (JSON-RPC Working Group, 2010), um formato de comunicação textual simples para executar processos e receber dados, para que usuários enviem transações e obtenham o estado atual da rede. Como essa interface é de baixo nível, existem bibliotecas, como a Web3.js, que abstraem as lógicas de acesso e o formato que os dados devem ser enviados para o nó, tornando simples a comunicação com uma dessas redes (Beyer, 2019).

A fim de se conectar à interface de um nó, é necessário informar um provedor, uma estrutura de dados que fornece um *link* que aponta para interface pública de um nó, para a biblioteca de abstração. A *Metamask* é uma carteira e, ao mesmo tempo, um exemplo de provedor, pois ela se conecta a um nó através da sua interface de baixo nível e expõe o *link* utilizado para conectar-se (Beyer, 2019).

5.2 Levantamento dos requisitos

Baseando-se em sistemas que permitem a doação de criptomoedas às instituições de caridade e em aplicações descentralizadas dos mais variados tipos, foram especificados os requisitos funcionais e não-funcionais da aplicação, representados nas tabelas 2 e 3.

Tabela 2. Requisitos funcionais da aplicação

Requisito funcional	Descrição
Cadastrar, editar e excluir instituições de caridade	Funcionalidade disponível somente ao administrador do contrato inteligente.
Visualizar instituições	Funcionalidade disponível a todos usuários. Visualização das instituições aptas a receber doações e seus dados.
Doar criptomoedas	Funcionalidade disponível a todos usuários. Permite a doação de qualquer quantidade de criptomoedas a uma instituição elegível.
Visualizar histórico de doações realizadas	Funcionalidade disponível a todos usuários. Permite visualizar cada doação feita, o valor doado e qual carteira que realizou a contribuição.
Visualizar quantidade total de doações	Funcionalidade disponível a todos usuários. Permite visualizar quantas doações cada instituição de caridade já recebeu.

Tabela 3. Requisitos não-funcionais da aplicação

Requisito não-funcional	Descrição
Disponibilidade	Os dados devem sempre estar disponíveis a qualquer momento e podem ser acessados por todos os usuários.
Usabilidade	Garante a facilidade na utilização da aplicação por meio de interfaces simples e de fácil compreensão.
Confiabilidade	Todos os dados devem ser armazenados na rede blockchain, de forma que fiquem seguros e sejam imutáveis.
Compatibilidade	Deve ser compatível com qualquer dispositivo que tenha acesso à internet e seja compatível com algum navegador *web* .

Após a especificação dos requisitos funcionais e não-funcionais, foi modelado o diagrama de caso de uso da aplicação representado na Figura 2.

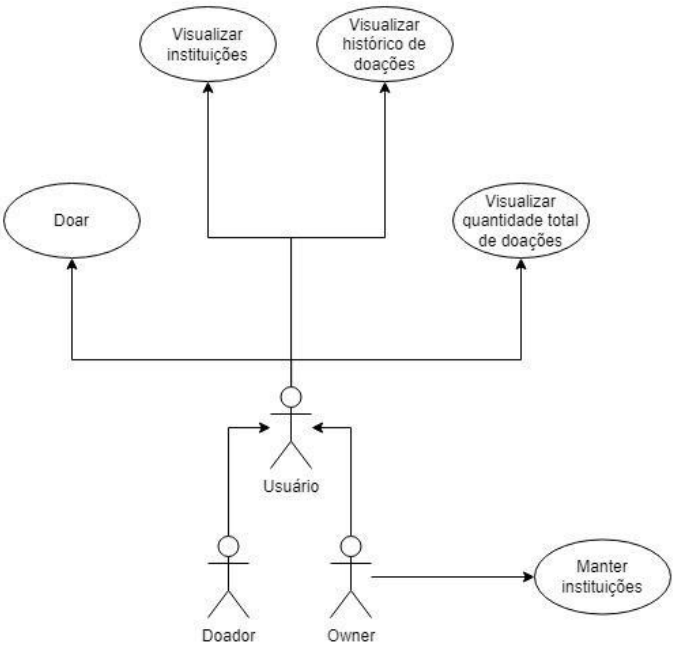


Figura 2. Diagrama de caso de uso da aplicação

Especificamente o caso de uso “Doar” é constituído de várias etapas que o usuário deve seguir para conseguir efetivar sua doação pela plataforma, por esse motivo, foi criado um diagrama para representar graficamente esse processo, a fim de torná-lo mais compreensível (Figura 3).

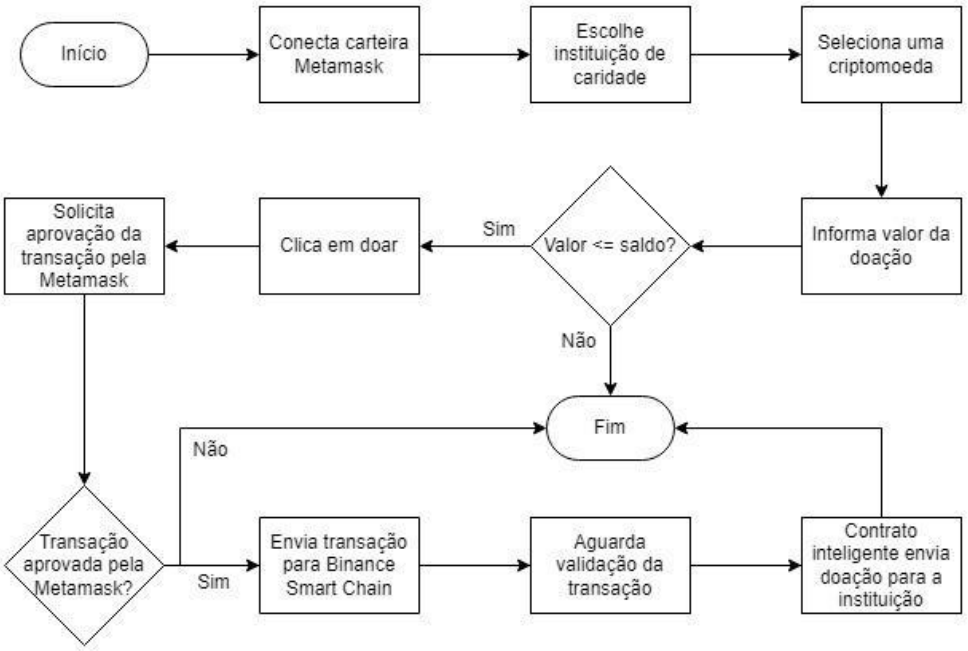


Figura 3. Fluxo do processo de doação

O processo da doação inicia-se a partir do momento que o usuário conecta a sua carteira *Metamask* na aplicação para poder ser utilizada e escolhe a instituição de caridade desejada, em sequência uma nova interface, que contém informações pertinentes à ONG, é exibida solicitando que o usuário selecione uma criptomoeda e informe o valor para ser enviado à instituição. Caso o valor da criptomoeda informado seja menor ou igual ao saldo dessa mesma moeda digital que possui em sua carteira, ele pode continuar o processo de doação clicando no botão, contudo, se a condição for falsa, o usuário não poderá doar e deverá escolher outra criptomoeda ou digitar um valor menor.

Ao clicar no botão doar, é necessário que o próprio usuário aprove manualmente a transação pela carteira, permitindo que parte do seu saldo da criptomoeda escolhida seja enviado para a instituição de caridade, por esse motivo, uma janela da *Metamask* é aberta solicitando a aprovação.

Se o usuário não aprovar a doação, todo o restante do processo é cancelado. Contudo, se a transação for aprovada, ela é enviada para a rede *Binance Smart Chain*, onde deve ser validada por algum nó e, ao ser validada, o contrato inteligente envia a doação para a instituição de caridade efetivando a contribuição com sucesso.

5.3 Modelagem dos dados

Visto que os contratos inteligentes são muito parecidos com as classes em linguagens orientadas a objetos, é possível utilizar-se do diagrama de classes da UML para representar a estrutura interna de um contrato, bem como a forma que os dados estão sendo armazenados (Rocha, Ducasse, 2018).

Uma das vantagens de utilizar esse diagrama para representá-lo é a possibilidade de modelar e especificar, não somente os dados, mas também as funções que o contrato inteligente implementa. Além disso, o padrão de modelagem UML é conhecido pela maioria dos desenvolvedores e engenheiros de *software*, facilitando a compreensão até mesmo das pessoas menos experientes no assunto (Rocha, Ducasse, 2018) (Figura 4).

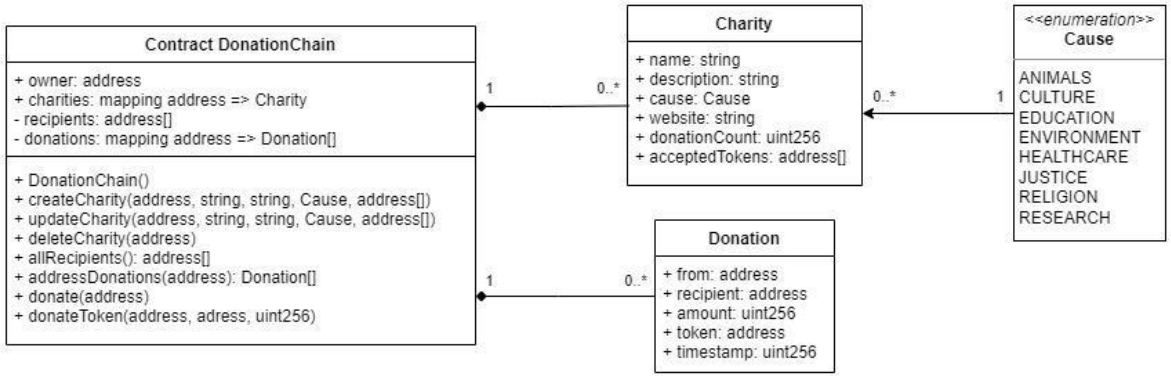


Figura 4. Diagrama de classes representado o contrato inteligente e seus dados

O contrato inteligente foi representado como uma classe, que armazena as instituições de caridade, as doações realizadas e quem é o administrador do contrato (*owner)*, e possui diversas funcionalidades implementadas. Por sua vez, as instituições de caridade e doações salvas na rede também foram retratadas como classes, pois cada uma delas possui dados específicos e estruturados.

5.4 Desenvolvimento da aplicação

Após a fase de levantamento dos requisitos e modelagem dos dados, foram planejados os incrementos a serem desenvolvidos baseando-se nos casos de uso da aplicação. Na Tabela 4 é retratado o caso de uso implementado, validado e entregue a cada incremento.

Tabela 4. Incremento produzido para cada caso de uso

Incremento	Caso de uso	Impacto
Incremento 1	Manter instituições	Contrato inteligente
Incremento *2*	Visualizar instituições	Contrato inteligente e interface gráfica
Incremento 3	Doar	Contrato inteligente e interface gráfica
Incremento 4	Visualizar quantidade total de doações e visualizar histórico de doações	Contrato inteligente e interface gráfica

Os incrementos foram implementados em duas etapas, primeiramente foi desenvolvida a respectiva funcionalidade no contrato e, logo após, foi criada a interface gráfica. Todas as funcionalidades do contrato inteligente foram desenvolvidas usando a linguagem de programação *Solidity*, na versão 0.8.7 e o ambiente de desenvolvimento *Remix* configurado com as credenciais do ambiente de testes da rede *Binance Smart Chain*. Por outro lado, a interface gráfica foi implementada com as linguagens HTML, CSS e *JavaScript* em conjunto com as bibliotecas *React*, *Mantine* e *Web3.js*.

5.5 Desenvolvimento do contrato inteligente

Primeiramente foi desenvolvido o caso de uso “Manter instituições”, sendo composto de 3 funcionalidades: cadastrar (Figura 5), alterar e remover. Para cada funcionalidade foi criada uma função que só pode ser executada pelo administrador do contrato e informando parâmetros válidos.

```
// Cadastra uma nova instituição de caridade
function createCharity(address recipient, string memory name, string memory description, Cause cause, string memory website, address[] memory acceptedTokens)
public isOwner() notExistCharity(recipient) isCharityValid(recipient, name, description, acceptedTokens) {
  charities[recipient] = Charity({
    name: name,
    description: description,
    cause: cause,
    website: website,
    acceptedTokens: acceptedTokens,
    donationCount: 0
  });

  recipients.push(recipient);
}
```

Figura 5. Código da função responsável por cadastrar instituições de caridade

A fim de realizar o cadastro ou alterar determinada instituição, são necessários os seguintes dados: nome, descrição, causa, quais moedas são aceitas e o endereço da carteira para onde as doações serão enviadas. As informações passadas são armazenadas em um *mapping*, uma estrutura formada por pares de chave-valor, sendo a chave o endereço da carteira da instituição e o valor a própria instituição de caridade. O endereço para onde as doações são enviadas também é adicionado a um vetor separado, denominado *recipients* (Figura 16 do Anexo 2).

Em seguida, para possibilitar a visualização das instituições de caridade cadastradas, foi necessário desenvolver a função *allRecipients*, responsável por retornar todos os endereços de recebimento, pois visto que a instituição é armazenada na estrutura *mapping*, só é possível obter dados de determinada ONG informando o endereço de sua carteira.

Logo após o caso de uso “Visualizar instituições”, foi implementado o processo de doação em si através das funções *donate* e *donateToken*. Elas são responsáveis por transferir o valor informado para o endereço da instituição de caridade, incrementar o contador de doações recebidas e salvar o registro da doação para manter o histórico de todas as contribuições que foram realizadas pelos usuários (Figura 6).

```
function donate(address payable recipient) public payable existCharity(recipient) senderNotRecipient() {
  recipient.transfer(msg.value);
  charities[recipient].donationCount += 1;

  Donation memory donation = Donation({
    from: msg.sender,
    recipient: recipient,
    amount: msg.value,
    token: address(0),
    timestamp: block.timestamp
  });
  donations[msg.sender].push(donation);
  donations[recipient].push(donation);
}
```

Figura 6. Código da função responsável por realizar a doação

Por fim, com o intuito de visualizar o histórico de doações que uma instituição de caridade específica recebeu ou obter as contribuições que o usuário já realizou pela aplicação, foi desenvolvida a função *addressDonations*, que retorna todas as doações de determinado endereço.

As doações realizadas são armazenadas num *mapping* de forma parecida a como as instituições de caridade são salvas no contrato. A chave desse mapeamento é o endereço da carteira que enviou ou recebeu a doação e o valor é um vetor de doações recebidas ou enviadas por aquele endereço (Figura 17 do Anexo 2), que contém informações sobre quem fez a doação, a instituição de caridade que recebeu, a data da contribuição e o valor e a criptomoeda utilizada.

5.6 Desenvolvimento da interface gráfica

No início do desenvolvimento da interface gráfica, primeiramente foi necessário configurar a biblioteca *Web3.js*, indicando qual o contrato inteligente que foi criado para controlar a aplicação, e desenvolver a conexão com a carteira *Metamask*. A fim de que um usuário utilize a interface gráfica para realizar doações, é essencial que ele conecte a carteira na aplicação, pois só assim a *Web3.js* consegue ter acesso aos dados desse usuário e solicitar a aprovação de transações. Na Figura 7 é mostrado o código da interface responsável por conectar a *Metamask* ao clicar no botão "Conectar Carteira".

```
const { ethereum: metamask } = window;

try {
  if (metamask) {
    const [addr] = await metamask.request({
      method: 'eth_requestAccounts',
    });
    await addOrSwitchNetwork();
    setAccount(addr);
    web3.setProvider(metamask);
    onSuccess({ account: addr });
  } else {
    throw new Error('Metamask não está instalada');
  }
} catch (e) {
  onError({
    message:
      e.code === 4001
        ? 'Usuário rejeitou a conexão com a carteira'
        : e.message,
  });
}
```

Figura 7. Código responsável por conectar a carteira na aplicação

Quando a *Metamask* é instalada no navegador, ela adiciona uma propriedade *ethereum* em todas as páginas *web* acessadas pelo usuário e, a partir desse objeto, as aplicações descentralizadas podem interagir com a carteira. Embora a propriedade tenha o mesmo nome da rede *Ethereum*, pois a carteira foi desenvolvida especificamente para essa *blockchain*, atualmente ela suporta qualquer rede baseada na EVM, incluindo a *Binance Smart Chain*.

Ao solicitar a conexão através do código *Javascript*, uma janela da *Metamask* é aberta para que o usuário confirme a conexão da carteira com a aplicação descentralizada (Figura 18 do Anexo 3).

Em seguida ao desenvolvimento da conexão com a carteira, foi implementado o caso de uso "Visualizar instituições" criando a interface inicial da aplicação que exibe todas as ONGs aptas a receber doações e é possível filtrá-las por causa defendida pelo menu lateral (Figura 19 do Anexo 3).

Como os dados das instituições são armazenados no contrato inteligente, foi utilizada a biblioteca *Web3.js* para obtê-los da *blockchain* e mostrá-los na tela (Figura 8).

```
const addresses = await contract.methods.allRecipients().call();
const charities = await Promise.all(
  addresses.map(addr =>
    contract.methods
      .charities(addr)
      .call()
      .then(charity => ({
        ... charity,
        recipient: addr,
      }))
  )
);

setCharities(charities);
```

Figura 8. Código que obtém as instituições de caridade aptas a receber doações

No seguinte incremento desenvolveu-se o processo de doação, portanto, foi criada a interface gráfica exibida após o usuário escolher uma instituição de caridade, onde é possível selecionar a moeda desejada, informar o valor e enviar a doação (Figura 20 do Anexo 3). O visual dessa interface e a disposição dos elementos foram fortemente inspirados na tela de doação da aplicação *Giveth.io*.

Ao clicar em "Doar", uma nova transação é criada, mas antes de ser enviada à rede, ela precisa ser aprovada pelo próprio usuário através da *Metamask* para confirmar sua autenticidade. Se for aprovada pela carteira, ela é enviada para a *Binance Smart Chain* e o usuário é notificado sobre o status atual da transação. O código encarregado de realizar esse processo é apresentado na Figura 9.

```
const weiAmount = web3.utils.toWei(amount.toFixed());

const transaction = contract.methods
  .donate(charity.recipient)
  .send({ from: account, value: weiAmount });

transaction
  .on('transactionHash', transactionHash ⇒
    onPending({ transactionHash })
  )
  .on('receipt', receipt ⇒ {
    onSuccess(receipt);
    setDonating(false);
  })
  .on('error', error ⇒ {
    onError(error);
    setDonating(false);
  });
```

Figura 9. Código que cria uma transação para executar o contrato e enviar a doação

Por fim, foi desenvolvido e entregue o último incremento contendo a implementação dos casos de uso "Visualizar quantidade total de doações" e "Visualizar histórico de doações".

Nesse incremento foi adicionada na interface a quantidade de doações que determinada instituição recebeu, para o usuário conseguir diferenciar quais as ONGs mais famosas e que recebem mais doações.

Além disso, para cumprir o segundo caso de uso, também foi adicionada uma listagem de doações recebidas pela instituição de caridade na mesma interface onde a contribuição é realizada, dessa forma o usuário pode visualizar as últimas doações realizadas pela aplicação e seus respectivos valores (Figura 21 do Anexo 3).

Com o intuito de permitir que o usuário sempre tenha acesso ao histórico de doações que ele mesmo fez, foi desenvolvida uma interface para listar as doações feitas pela carteira conectada (Figura 22 do Anexo 3).

5.7 Testes

A fim de garantir a integridade e o funcionamento correto do contrato inteligente, foram desenvolvidos testes unitários de acordo com os requisitos funcionais da aplicação, também evitando possíveis falhas durante o uso da aplicação pelos usuários.

Os testes unitários do contrato inteligente foram implementados utilizando a linguagem *JavaScript* em conjunto com a biblioteca de testes *Mocha* (Mocha, 2022). Apesar dos contratos serem desenvolvidos utilizando a linguagem *Solidity*, eles podem ser testados

com *JavaScript*, porque após serem compilados, é possível executá-los normalmente em qualquer ambiente *web*.

A Figura 23 do Anexo 4 mostra a execução dos testes unitários do contrato inteligente implementados com a biblioteca *Mocha*. Por meio do relatório apresentado é possível visualizar que cada funcionalidade foi testada individualmente e todos os testes passaram sem apresentar qualquer erro, devido aos símbolos verdes no lado esquerdo.

Visto que os contratos inteligentes precisam ser executados para serem testados, é necessário simular uma rede *blockchain* local no dispositivo de quem solicitou a execução dos testes unitários. Tal simulação é feita pela biblioteca *Mocha*, que fornece um ambiente propício para o contrato inteligente, contudo, as funcionalidades dele demoram um certo tempo para serem testadas, pois transações são criadas e precisam ser validadas nessa *blockchain* local.

6. Conclusão

O presente trabalho apresentou o desenvolvimento de uma aplicação descentralizada que visa arrecadar fundos em moedas digitais para instituições de caridade, incentivando principalmente as pessoas detentoras de criptomoedas a realizarem doações a ONGs por meio dessa plataforma confiável.

A aplicação foi implementada seguindo todos os requisitos descritos neste trabalho, visto que é possível cadastrar as instituições de caridades, enviar doações em criptomoedas e visualizar informações relevantes sobre as doações já realizadas pela plataforma. A plataforma desenvolvida cumpriu todas as regras necessárias para ser considerada uma aplicação descentralizada e, dado que ela foi planejada para ter uma alta disponibilidade e confiabilidade, o contrato inteligente, responsável por manipular e armazenar os dados, foi desenvolvido e publicado utilizando os recursos da rede *Binance Smart Chain*.

Como trabalhos futuros é possível desenvolver uma evolução da aplicação proposta contendo uma nova interface gráfica para gerenciar as instituições de caridade, ou seja, seria permitido cadastrar, alterar ou até mesmo remover as ONGs aptas a receberem doações pela plataforma, auxiliando o administrador do contrato nesta tarefa.

Para o desenvolvimento deste trabalho foi utilizado o conhecimento adquirido em diversas disciplinas ministradas no decorrer do Curso Superior de Análise e Desenvolvimento de Sistemas, destacando as matérias de Desenvolvimento Web, Engenharia de Software e Programação Orientada a Objetos. Além disso, foram adquiridos diversos conhecimentos extras sobre o funcionamento da tecnologia *blockchain* e de redes que a implementam, como a *Ethereum* e a *Binance Smart Chain*, utilização da linguagem de programação *Solidity* e desenvolvimento de aplicações descentralizadas.

Referências

Abijaude, J. W., Greve, F. and Sobreira, P. L. (2021). *Blockchain e Contratos Inteligentes para Aplicações em IoT, Uma Abordagem Prática*. Sociedade Brasileira de Computação.

Beyer, S. (2019). *What is Web3.js? A Detailed Guide*. Disponível em: <https://www.mycryptopedia.com/what-is-web3-js-a-detailed-guide/>. Acesso em: 15 nov. 2021.

Binance (2021). *Binance Smart Chain*. Disponível em: <https://docs.binance.org/faq/bsc/bsc.html#can-you-tell-more-about-proof-of-staked-authorityposa-what-is-it>. Acesso em: 8 jan. 2022.

Bison Trails (2021). *Guide to Binance Smart Chain*. Disponível em: <https://bisontrails.co/guide-to-bsc>. Acesso em: 8 jan. 2022.

CoinMarketCap (2021). *Gráficos Globais de Criptomoedas: Total Cryptocurrency Market Cap*. Disponível em: <https://coinmarketcap.com/pt-br/charts>. Acesso em: 27 jul. 2021.

Copes. F. (2019). *The React Handbook*. Disponível em: <https://www.freecodecamp.org/news/the-react-handbook-b71c27b0a795>. Acesso em: 15 nov. 2021.

Giveth (2021). *Giveth Docs.* Disponível em: <https://docs.giveth.io>. Acesso em: 28 jul. 2021.

Greve, F. et al (2018). *Blockchain e a Revolução do Consenso sob Demanda*. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - Minicursos.

Dannen, C. (2017). *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners.* Apress, 1ª edição.

Ethereum (2021). *Solidity.* Disponível em: <https://docs.soliditylang.org/en/v0.8.9/index.html>. Acesso em: 31 out. 2021.

Ethereum (2021). *Introduction to dapps.* Disponível em: <https://ethereum.org/pt-br/developers/docs/dapps>. Acesso em: 15 nov. 2021.

JSON-RPC Working Group (2010). *JSON-RPC 2.0 Specification.* Disponível em: <https://www.jsonrpc.org/specification>. Acesso em: 24 jan. 2022.

Fowler, M. (2003). *UML Distilled: a brief guide to the standard object modeling language.* Addison-Wesley Longman Publishing, 3ª edição.

Hassan, N., Jain, N. and Chandna, V. K. (2018). *BLOCKCHAIN, CRYPTOCURRENCY AND BITCOIN.* Disponível em: <https://www.researchgate.net/publication/334279715_BLOCKCHAIN_CRYPTOCURRENCY_AND_BITCOIN>. Acesso em: 11 jul. 2021.

Laurence, T. (2017). *Blockchain for Dummies.* For Dummies, 1ª edição.

Mantine (2021). *Mantine: A fully featured React components library.* Disponível em: <https://mantine.dev>. Acesso em: 15 nov. 2021.

MDN Web Docs (2021). *HTML: HyperText Markup Language.* Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTML>. Acesso em: 15 nov. 2021.

MDN Web Docs (2021). *CSS: Cascading Style Sheets.* Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS>. Acesso em: 15 nov. 2021.

MDN Web Docs (2021). *JavaScript.* Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Acesso em: 15 nov. 2021.

Mobiliza, ReosPartners (2020). *Impacto da COVID-19 nas OSCs brasileiras: da resposta imediata à resiliência.* Disponível em: <https://mcusercontent.com/d468d6493b34f50f5ac335f91/files/4cb971d3-11db-4790-8262-48a3803985e2/sumario_estudo_ocs_3.pdf>. Acesso em: 27 jul. 2021.

Mocha (2022). *Mocha: Simple, flexible, fun.* Disponível em: <https://mochajs.org>. Acesso em: 19 jan. 2022.

Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System.* Disponível em: <https://bitcoin.org/bitcoin.pdf>. Acesso em: 23 jan. 2022.

Peyrott, S. (2017). *AN INTRODUCTION TO ETHEREUM AND SMART CONTRACTS.* Disponível em: <https://assets.ctfassets.net/2ntc334xpx65/42fINJjatOKiG6qsQQAyc0/8b63e552f4cfef313f579b8e9c9154b5/intro-to-ethereum.pdf>. Acesso em: 7 jan. 2022.

Pressman, R. S. (2011). *Engenharia de Software: Modelos de Processos.* AMGH, 8ª edição.

Remix. (2021). *Welcome to Remix's documentation!* Disponível em: <https://remix-ide.readthedocs.io/en/latest>. Acesso em: 31 out. 2021.

Rocha, H. and Ducasse S. (2018). *Preliminary Steps Towards Modeling Blockchain Oriented Software.* Disponível em: <https://hal.inria.fr/hal-01831046/document>. Acesso em: 9 jan. 2022.

Rugdoc. (2021). *Proof of Stake Authority (PoSA).* Disponível em: <https://wiki.rugdoc.io/docs/proof-of-stake-authority-posa>. Acesso em: 8 jan. 2022.

Singhal, B., Dhameja, G. and Panda, P. (2018). *Beginning Blockchain: a beginner's guide to building blockchain solutions.* Apress, 1ª edição.

Souza, F. (2020). *Funções hash ou hashing?* Disponível em: <https://medium.com/prognosys/fun%C3%A7%C3%B5es-hash-ou-hashing-b2c90ac5c398>. Acesso em: 8 jan. 2022.

Stallings, W. (2014). *Criptografia e segurança de redes: princípios e práticas.* Pearson Education do Brasil, 6ª edição.

The Giving Block. (2018). *The Giving Block.* Disponível em: <https://www.thegivingblock.com>. Acesso em: 28 jul. 2021.

Vadapalli, R. (2020). *BLOCKCHAIN FUNDAMENTALS TEXT BOOK.* Blockchainprep UAE, 1ª edição.

Anexo 1

Este anexo contém diagramas que exemplificam a utilização de funções *hash* e detalham conceitos presentes na tecnologia *blockchain*.

Na Figura 10 são apresentados alguns exemplos de mensagens processadas por uma determinada função *hash* e o *hash* gerado para as respectivas entradas.

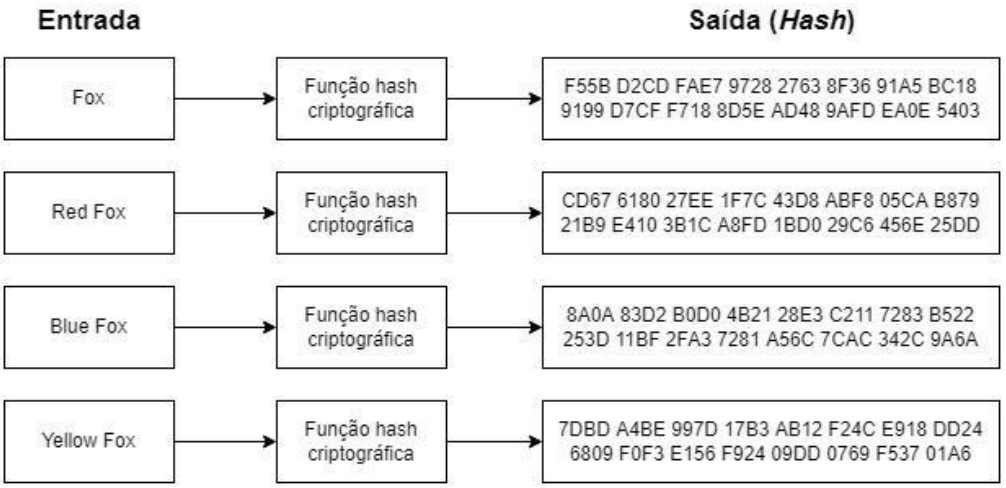


Figura 10. Exemplos de mensagens processadas por determinada função *hash*

A Figura 11 retrata o processo de inclusão de uma transação no novo bloco assim que ela é validada com sucesso por algum nó da *blockchain*.

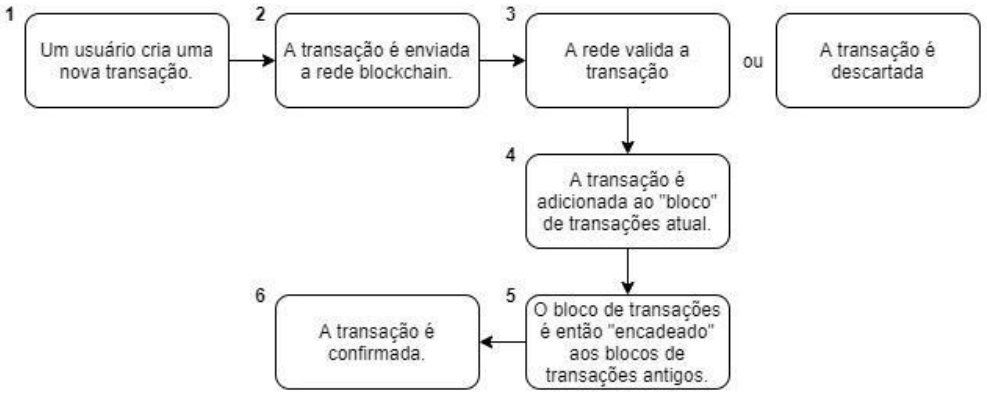


Figura 11. Como uma *blockchain* funciona (adaptado de Laurence, 2017)

A Figura 12 mostra como os blocos da *blockchain* são encadeados "matematicamente" possibilitando a visualização do histórico dos dados armazenados na rede.

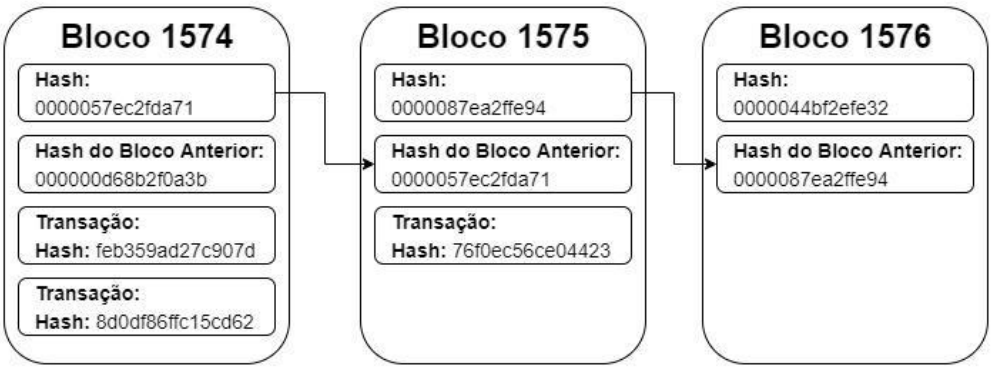


Figura 12. Encadeamento de blocos por meio dos *hashes* gerados (adaptado de Vadapalli, 2020)

Na Figura 13 é exemplificado as modificações no estado da *blockchain* causadas pelas transações.

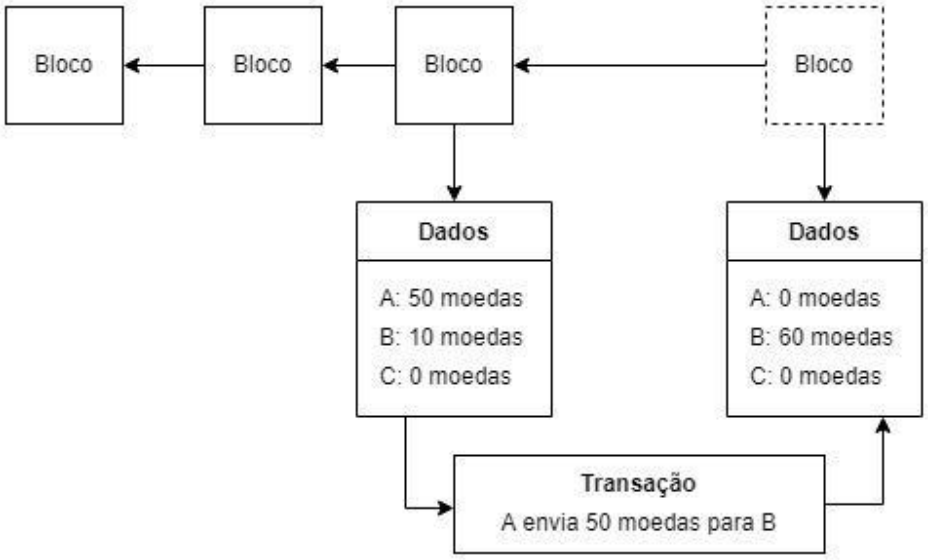


Figura 13. Mudança de estado em uma *blockchain* (adaptado de Peyrott, 2017)

Anexo 2

Este anexo apresenta o ambiente de desenvolvimento *Remix* e códigos de contratos inteligentes escritos utilizando a linguagem de programação *Solidity*.

A Figura 14 retrata um exemplo de contrato inteligente que armazena e manipula um contador ao ser executado.

```
pragma solidity ^0.8.10;

contract Contador {
    uint public cont;

    // Function to get the current count
    // Função para obter o valor atual do contador
    function get() public view returns (uint) {
        return cont;
    }

    // Função para incrementar o contador em 1
    function inc() public {
        cont += 1;
    }

    // Função para decrementar o contador em 1
    function dec() public {
        cont -= 1;
    }
}
```

Figura 14. Exemplo de contrato inteligente desenvolvido com Solidity

A Figura 15 apresenta a interface inicial do ambiente de desenvolvimento Remix ao ser acessado pela primeira vez.

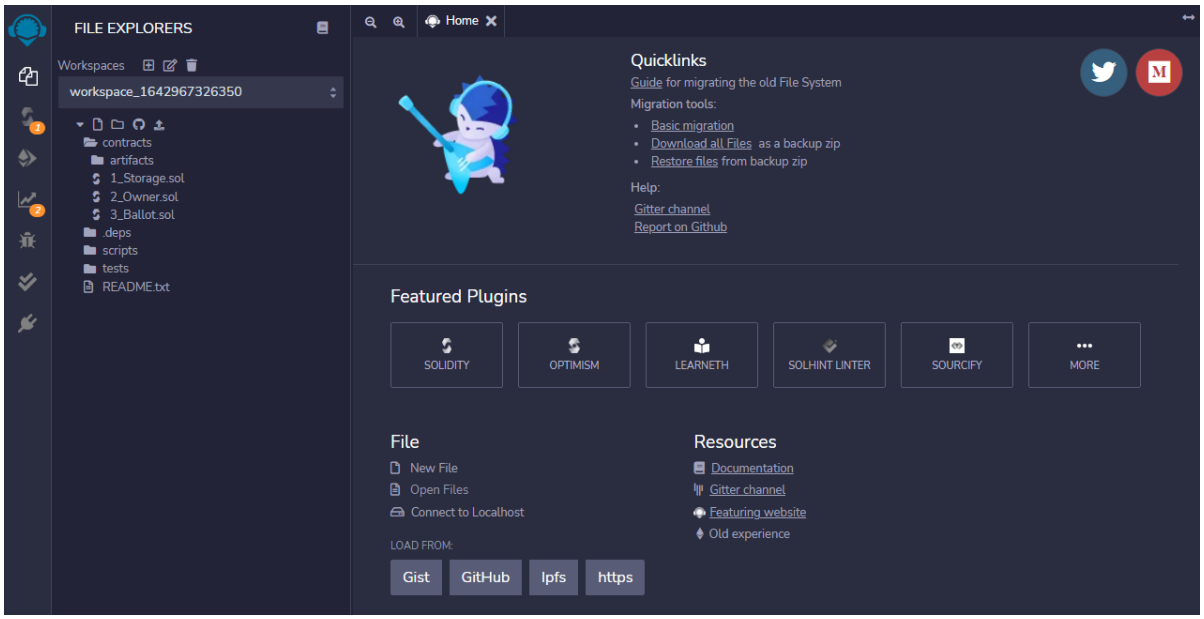


Figura 15. Interface inicial do ambiente de desenvolvimento Remix

Na Figura 16 é mostrado o formato que os dados das instituições de caridade são armazenados no contrato inteligente.

```
struct Charity {
  string name;
  string description;
  Cause cause;
  string website;
  uint256 donationCount;
  address[] acceptedTokens;
}

mapping(address => Charity) public charities;
address[] private recipients;
```

Figura 16. Formato que as instituições são armazenadas no contrato inteligente

Na Figura 17 é mostrado o formato que os dados das instituições de caridade são armazenados no contrato inteligente.

```
struct Donation {
  address from;
  address recipient;
  uint256 amount;
  address token;
  uint256 timestamp;
}

mapping(address => Donation[]) private donations;
```

Figura 17. Histórico de doações armazenado no contrato

Anexo 3

Este anexo contém imagens da carteira *Metamask* e das interfaces gráficas da aplicação desenvolvida.

Na Figura 18 é apresentada a janela da *Metamask* aberta quando o usuário solicita a conexão com a carteira.

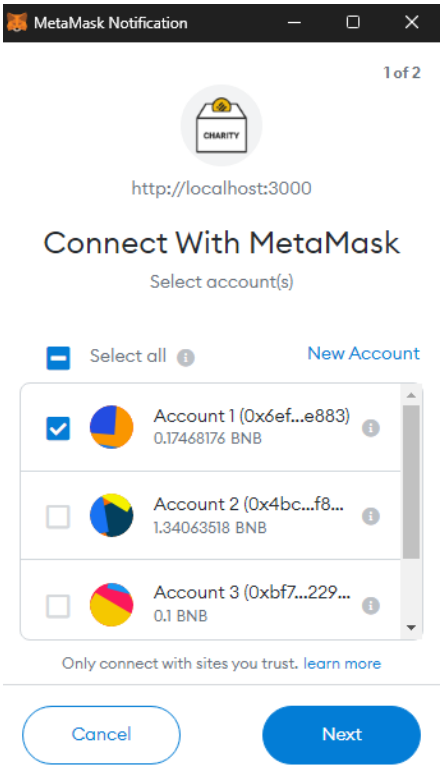


Figura 18. Janela da *Metamask* aberta ao solicitar a conexão

A Figura 19 apresenta a interface gráfica inicial da aplicação com a lista de instituições de caridade aptas a receber doações pela plataforma.

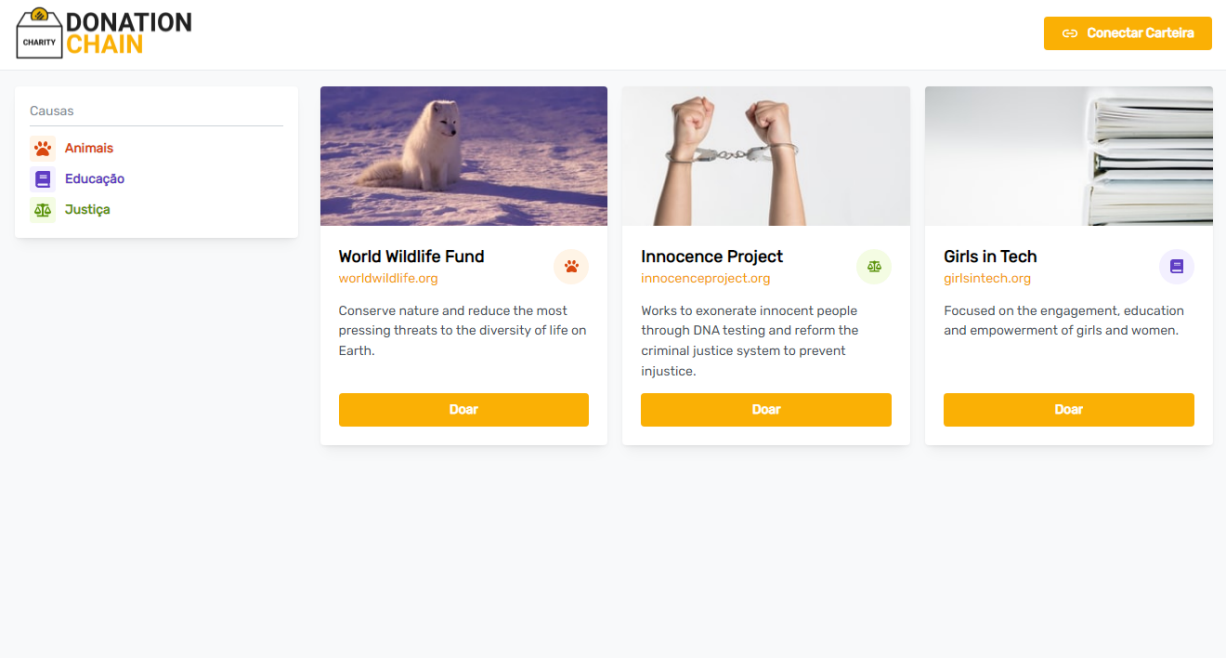


Figura 19. Interface inicial da aplicação que exibe a lista de instituições de caridade

A Figura 20 apresenta a interface gráfica exibida ao escolher uma instituição para doar criptomoedas.

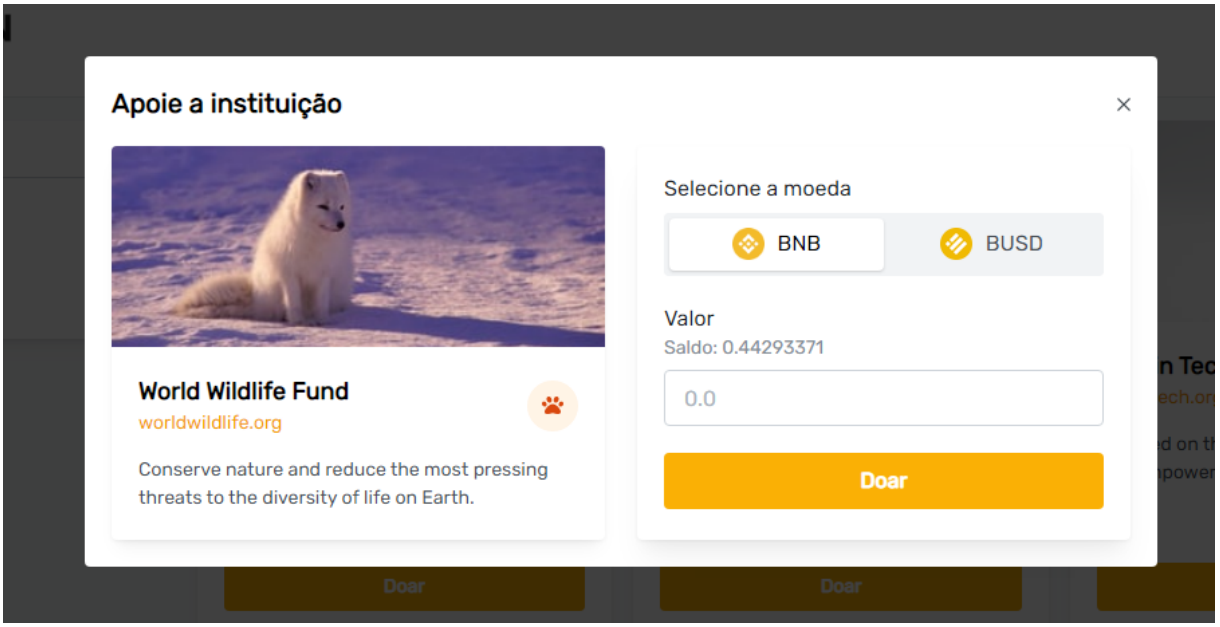


Figura 20. Interface para doar criptomoedas à ONG escolhida

Na Figura 21 é exibida a interface gráfica de doação com o histórico de doações recebidas pela instituição de caridade.

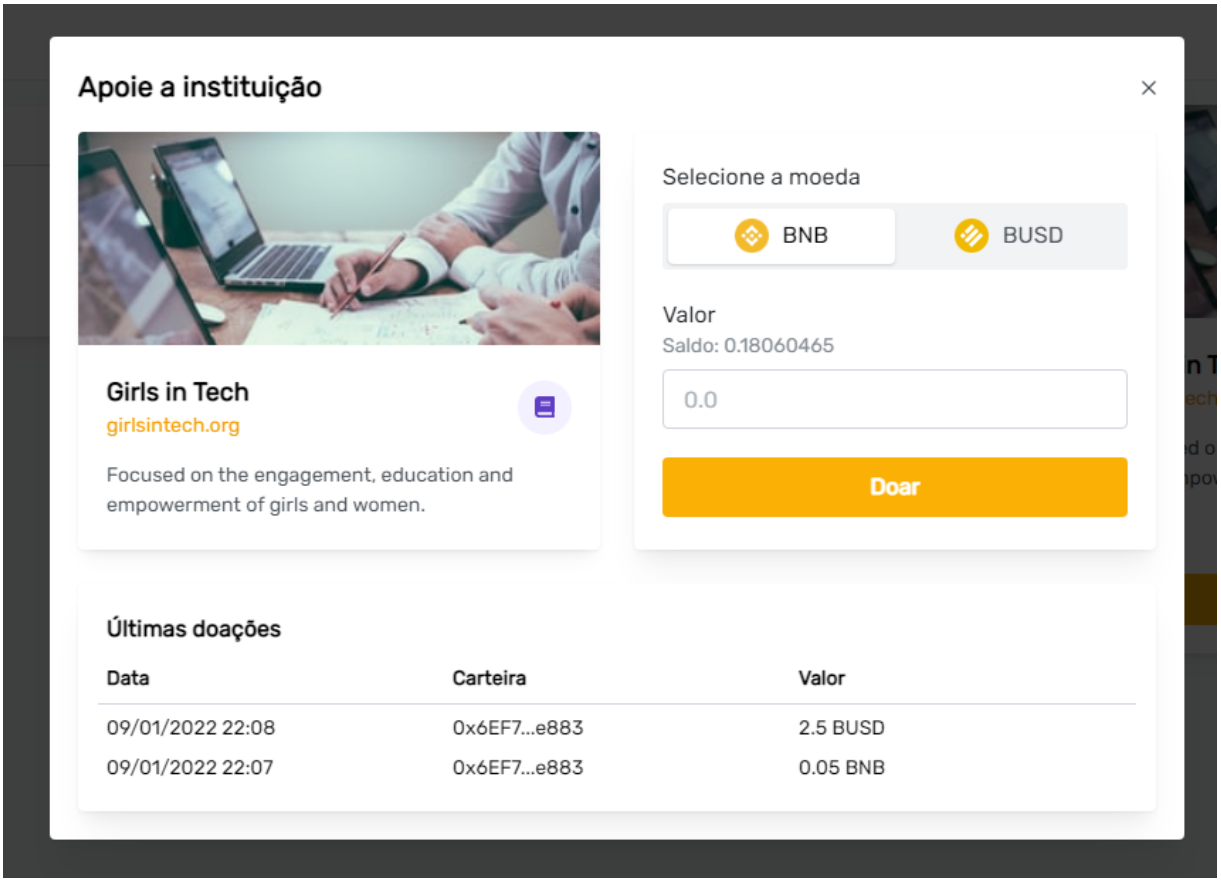


Figura 21. Interface de doação com últimas doações recebidas pela instituição de caridade

Na Figura 22 é apresentada a interface gráfica de detalhes da carteira conectada com a histórico de doações realizadas pelo próprio usuário.

Figura 22. Lista de doações realizadas pela carteira conectada à aplicação

Anexo 4

Este anexo, por meio da Figura 23, apresenta o relatório produzido após a execução dos testes unitários do contrato inteligente, que foram implementados utilizando a biblioteca *Mocha*.

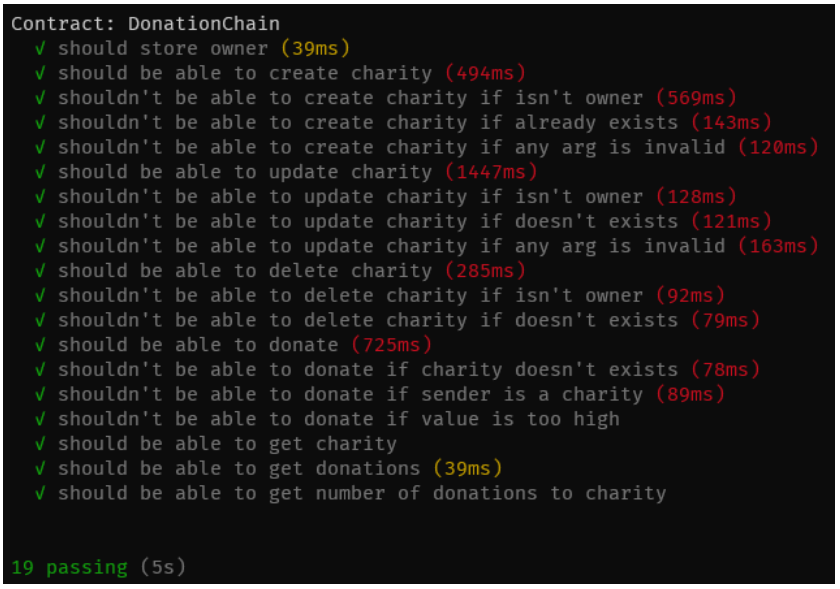


Figura 23. Relatório produzido após a execução dos testes unitários do contrato inteligente