UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

MATHEUS MAIA AMORIM
Orientador: Prof. Dr. Carlos Frederico Marcelo da Cunha Cavalcanti

IMPLEMENTAÇÃO DE CONTRATO INTELIGENTE EM SOLIDITY PARA GESTÃO DE IDENTIDADE DESCENTRALIZADA (DID)

Ouro Preto, MG
2024

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

MATHEUS MAIA AMORIM

IMPLEMENTAÇÃO DE CONTRATO INTELIGENTE EM SOLIDITY PARA GESTÃO DE IDENTIDADE DESCENTRALIZADA (DID)

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Carlos Frederico Marcelo da Cunha Cavalcanti

Ouro Preto, MG
2024

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Matheus Maia Amorim

IMPLEMENTAÇÃO DE CONTRATO INTELIGENTE EM SOLIDITY PARA GESTÃO DE IDENTIDADE DESCENTRALIZADA (DID)

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 21 de Outubro de 2024.

Membros da banca

Carlos Frederico M. da Cunha Cavalcanti (Orientador) - Doutor - Universidade Federal de Ouro Preto
Ricardo Augusto Rabelo Oliveira (Examinador) - Doutor - Universidade Federal de Ouro Preto
Fernando Cortez Sica (Examinador) - Doutor - Universidade Federal de Ouro Preto

Carlos Frederico M. da Cunha Cavalcanti, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 12/11/2024.

Documento assinado eletronicamente por **Carlos Frederico Marcelo da Cunha Cavalcanti**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 13/11/2024, às 11:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0795432** e o código CRC **0FB7468D**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Resumo

A utilização de identidades digitais está se tornando um tema cada vez mais relevante no contexto atual. Conforme grande parte das atividades realizadas no mundo todo estão migrando para o ambiente online, é necessário uma garantia de que as informações de cada usuários não sejam expostas. Nesse cenário surge uma proposta promissora para gerenciamento dos dados pessoais, a gestão de identidade descentralizada, que utiliza o *blockchain Ethereum* para manter e distribuir de forma segura essas informações. Tendo em vista essa proposta, este trabalho tem como objetivo a arquitetura e a implementação de contrato inteligente para garantir a autenticidade das identidades registradas e a exploração de soluções para desafios específicos na implementação de sistemas DID, Identidade Digital Descentralizada. Para a construção proposta foram realizadas diversas pesquisas de formas de cadastro de usuário, análise de tecnologias existentes e requisitos que possam tornar o contrato íntegro. Os resultados apresentados demonstram uma eficiência do contrato inteligente, promovendo um ambiente seguro e flexível para gerenciamento e armazenamento de dados. Com as pesquisas realizadas e os resultados apresentados, é possível concluir que a abordagem utilizada, utilizando a linguagem de programação *Solidity* e o *blockchain Ethereum* é bastante promissora para um ambiente de gestão de identidades descentralizadas, permitindo uma flexibilidade para adaptações específicas e proporcionando uma garantia de segurança. O estudo destaca a importância de uma identidade digital descentralizada e aponta os possíveis trabalhos futuros, ampliando assim a utilidade da solução.

Palavras-chave: *Blockchain*, Contrato Inteligente, Solidity, Identidade Digital Descentralizada.

Abstract

The use of digital identities is becoming an increasingly relevant topic in the current context. As a large part of activities around the world are moving online, there is a need to ensure that users' information is not exposed. In this scenario, a very interesting proposal for the management of personal data emerges, decentralized identity management, which uses the Ethereum blockchain to securely maintain and distribute this information. With this proposal in mind, this work aims at the architecture and implementation of a smart contract to guarantee the authenticity of the registered identities and the exploration of solutions for specific challenges in the implementation of Decentralized Digital identity -DID- systems. For the proposed construction, various researches were conducted on user registration methods, analysis of existing technologies, and requirements that could make the contract integral. The results presented demonstrate the efficiency of the smart contract, promoting a secure and flexible environment for the management and storage of data. With the research conducted and the results presented, it is possible to conclude that the approach used, utilizing the Solidity programming language and the Ethereum blockchain, is very promising for an environment of decentralized identity management, allowing flexibility for specific adaptations and providing a guarantee of security. The study highlights the importance of decentralized digital identity and points out possible future works, thus expanding the utility of the solution.

Keywords: Blockchain. Smart Contract. Solidity. Decentralized Digital identity.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABNT	Associação Brasileira de Normas Técnicas
DECOM	Departamento de Computação
DID	Identidade Digital Descentralizada
DLT	*Distributed Ledger Technology*
ETH	*Ether*
EVM	Máquina Virtual *Ethereum*
IdM	*Identity Management*
PKI	*Public Key Infrastructure*
RG	Registro Geral
SSI	*Self-Sovereign Identity*
SSL	*Security Sockets Layer*
TLS	*Transport Layer Security*
UFOP	Universidade Federal de Ouro Preto

Lista de Símbolos

♢	Ether
Ξ	Letra grega Xi

Sumário

1 Introdução

A identidade digital (ID) é uma representação virtual da identidade de uma entidade neste mundo atual interconectado. Uma identidade digital engloba dados pessoais, como nome, data de nascimento, telefone, e-mail, endereços e demais outras características que podem ser utilizadas para autenticações e interações na Internet. Conforme as atividades do cotidiano começam a migrar para um ambiente online, a garantia de que essas informações pessoais não serão expostas torna-se fundamental, uma vez que, caso esses dados sejam visíveis, a prevenção contra fraudes e a proteção da privacidade não serão concretas. Elementos como credenciais de login, autenticação de dois fatores e a criptografia exercem um papel importante na proteção da identidade digital. No entanto, o desafio existente é descobrir um equilíbrio entre a utilidade digital e as formas de proteção de acordo com a evolução da identidade digital.

A identidade descentralizada (DID) emerge como uma resposta inovadora e promissora para os desafios relacionados à segurança e privacidade na administração de informações pessoais. A crescente interconexão digital e a consequente necessidade de proteger dados sensíveis tornam a gestão de identidade uma preocupação crucial. A descentralização proporcionada por contratos inteligentes representa não apenas uma solução atual, mas também uma abordagem futurista para lidar com questões de segurança e privacidade. Ao descentralizar os dados não é mais necessário confiar em uma autoridade central, com isso, há uma redução de riscos de falhas e vazamento de dados. Além de dar autonomia aos usuários para como, quando, onde e com quem compartilhar suas informações. Desse modo a identidade descentralizada atende às questões de segurança da atualidade e revoluciona como gerenciamos nossa presença no mundo online.

A tecnologia *blockchain*, conhecida por sua segurança e transparência, aliada aos contratos inteligentes, oferece um ambiente propício para a gestão de identidade. *Solidity*, a linguagem de programação predominante para contratos inteligentes, torna-se a escolha natural, proporcionando uma base sólida para a implementação prática. Roubos de identidade e violações de privacidade destacam a urgência de encontrar soluções inovadoras. A gestão de identidade descentralizada surge como uma alternativa capaz de superar os desafios existentes, e a implementação prática destas soluções pode oferecer *insights* valiosos sobre benefícios e desafios associados.

1.1 Justificativa

Com a crescente evolução da identidade digital, ter um controle das informações pessoais em um mundo conectado se torna crucial, já que existem diversas adversidades que comprometem a privacidade e segurança dos usuários. Existem diversas soluções para controlar os problemas com dados sensíveis na Internet e este trabalho procura explorar motivações para a escolha de uma abordagem de um sistema personalizado em *Solidity*, no *blockchain Ethereum*, apontando os

benefícios de uma implementação própria como a flexibilidade, adaptabilidade e controle destas informações.

O desenvolvimento de um contrato inteligente com *Solidity*, permite um maior controle sobre a lógica a ser implementada no sistema de identidade. Assim permitindo a utilização de regras específicas, o que pode levar a adaptações para tornar o contrato único e não generalizado. A linguagem foi desenvolvida para utilizar os padrões do *Ethereum*, promovendo assim a interoperabilidade, além de que, é possível usufruir da sólida segurança criptográfica e imutabilidade do *blockchain*.

Portanto, o desenvolvimento de um contrato inteligente de gestão de identidade descentralizada utilizando o *Solidity* e o *Ethereum*, é tido como uma boa abordagem levando em consideração a flexibilidade de mudanças à regras específicas e a segurança que a *blockchain* proporciona.

1.2 Objetivos

O objetivo principal deste trabalho é a implementação de um sistema simples de Gestão de Identidade Descentralizada.

E como objetivos específicos é tido:

- Arquitetura e implementação de um contrato inteligente para gestão de identidade descentralizada em *Solidity*;
- Examinar como o contrato inteligente pode ser utilizado para ambientes à prova de falsificação e garantir a autenticidade das identidades registradas;
- Identificar os desafios específicos e explorar soluções para a implementação correta do sistema de gestão DID em *Solidity*.

1.3 Organização do Trabalho

A estrutura deste trabalho é organizado da seguinte forma. No capítulo 2, são informados os principais conceitos que dizem respeito sobre Identidade Digital, Contratos Inteligentes, *blockchain*, *Ethereum*, *Solidity*, *Javascript* e *HardHat* . No capítulo 3, apresentam-se os requisitos essenciais e a estrutura para a realização deste trabalho. No capítulo 4, apresentam-se os resultados finais e discussões da solução desenvolvida. No capítulo 5, são apresentadas as as considerações finais e formas para progressão deste trabalho.

2 Revisão Bibliográfica

Será abordado nesse capítulo os trabalhos relacionados ao campo de interesse deste trabalho e apresemtar os referênciais teóricos que auxiliam em sua concepção.

2.1 Fundamentação Teórica

Nesta seção é apresentadada uma revisão geral da literatura atual e das definições fundamentais para a compreensão desse trabalho.

2.1.1 Identidade Digital

A identidade de uma entidade, que pode ser uma pessoa, objeto ou instituição, abrange todas as características que a definem em um determinado ambiente. A individualização de uma pessoa ocorre por meio de várias formas, como documentos de identificação (como RG), endereço de e-mail e credenciais de acesso (como logins e senhas). Além disso, essa entidade possui uma variedade de atributos, que podem incluir características físicas, biométricas e gênero, bem como credenciais que autorizam a realização de determinadas atividades profissionais.

Em um artigo Kim Cameron, arquiteto-chefe de identidades da *Microsoft*, argumenta que a internet foi concebida sem uma camada de identificação embutida (Siqueira; Conceição; Rocha, 2021). Em outras palavras, não há, por padrão, mecanismos que permitam aos usuários verificar a autenticidade de websites, nem que permitam websites verificar seus usuários. Essa lacuna resultou na criação de mecanismos suplementares para abordar essa ausência:

- Foi implementada uma solução de infraestrutura de de chaves públicas (*Public Key Infrastructure* ou PKI). Isso viabiliza a criação de autoridades certificadoras de validar certificados SSL/TLS utilizados pelos websites. Esses certificados tornam-se confiáveis para os usuários quando suas cadeias cerificadoras podem ser verificadas, conforme identificado pelos navegadores (Siqueira; Conceição; Rocha, 2021).
- Outra abordagem envolve a criação de mecanismos de autenticação baseados em credenciais, como nomes de usuários e senhas. Nesse caso, os usuários passam por um processo de cadastro prévio para estabelecer uma "identidade digital"especifíca, que pode ser solicitada durante a navegação para verificar a identidade (Siqueira; Conceição; Rocha, 2021).

No entanto, essas soluções centralizadas têm desvantagens, tornado os mecanismos de identidade digital tanto dispendiosos quanto pouco práticos. Além disso, geram pontos centrais

de vulnerabilidade, tornando essas entidades confiáveis alvos preferenciais para ataques de segurança.

É importante notar que a utilidade dessas estruturas de identidade digital vai além da simples navegação em sites da internet. O recente avanço na digitalização de informações demanda mecanismos de identificação mais abrangentes.

2.1.2 Identidade Digital Autossoberana

Há vinte e cinco anos, Peter Steiner trouxe à tona a famosa afirmação em um *cartoon*, "Na Internet, ninguém sabe que você é um cachorro", e surpreendentemente, essa observação continua relevante, destacando o persistente desafio de identificar pessoas online (Nakamura *et al.*, 2019).

Figura 2.1 – Cartoon Peter Steiner

Atualmente, estamos distantes das expectativas de diretórios públicos dos anos 70, impulsionadas pela criptografia de chave pública, ou do grande esquema de certificação hierárquica prevista nos anos 80. O campo do gerenciamento de identidades (IdM) na internet ainda lida com o que Cameron descreveu há mais de uma década como uma "colcha de retalhos de identidades únicas", caracterizada por diferentes sistemas IdM restritos a domínios específicos e com pouca interação entre eles.

Os modelos centralizados de IdM enfrentam desafios significativos devido às crescentes regulamentações relacionadas a violações de dados, resultando um danos à reputação, fraude de identidade e, acima de tudo, perda de privacidade para os indivíduos envolvidos. Esses eventos recorrentes destacam a falta de controle e gestão que os usuários têm sobre suas identidades digitais.

Para abordar essas questões, estão em andamento pesquisas sobre alternativas ao IdM. Iniciativas como a Estratégia Nacional dos Estados Unidos para Identidades Confiáveis no Ciberespaço (NSTIC) buscam acelerar o desenvolvimento de tecnologias que aumentem a confiança nas transações online. O ID2020, por sua vez, busca aproveitar tecnologias digitais emergentes

para ampliar o alcance de identidades legais, alinhando-se com metas das Nações Unidas de fornecer identidade digital para todos até 2030, incluindo registro de nascimento (Nakamura *et al.*, 2019).

O surgimento de moedas virtuais, como o *BitCoin* e o *Ethereum*, provocou uma reavaliação da identidade digital, devido o uso de Tecnologia de Livro-Razão Distribuído (DLT), que dispensa uma autoridade central para validar transações por meio de sua criptografia natural. Isso possibilita uma rede globalmente descentralizada capaz de chegar a um consenso sobre o estado atual das transações, proporcionando benefícios significativos para a aplicação do DLT ao IdM, como descentralização, inviolabilidade, inclusividade, redução de custos e controle do usuário sobre identificadores digitais, mesmo em caso de perda de acesso a serviços especifícos de provedores de identidade.

2.1.3 *Blockchain*

Um *blockchain* é um banco de dados distribuído que registra todas as transações realizadas na rede *blockchain*. Essa base de dados é replicada e compartilhada entre os participantes da rede. A característica principal do *blockchain* é possibilitar que participantes não confiáveis se comuniquem e realizem transações de maneira segura, eliminando a necessidade de uma entidade confiável intermediária.

O *blockchain* é essencialmente uma lista ordenada de blocos, sendo cada bloco identificado por seu hash criptográfico. Cada bloco referencia o bloco anterior, formando assim uma cadeia de blocos. Cada bloco contém um conjunto específico de transações, e uma vez que um bloco é criado e adicionado ao *blockchain*, as transações contidas nele tornam-se imutáveis e irrevogáveis. Essa imutabilidade é crucial para preservar a integridade das transações e prevenir problemas de gastos duplos (Alharby; Moorsel, 2017).

As criptomoedas representam a primeira geração da tecnologia *blockchain*, sendo essencialmente moedas digitais fundamentadas em técnicas criptográficas e redes ponto-a-ponto. O *Bitcoin* é o exemplo inaugural e mais proeminente nesse contexto. Na rede do *Bitcoin*, as transações são verificadas por especializados chamados de mineradores, os quais, ao resolverem um quebra-cabeça matemático, geram e propagam novos blocos de transações.

Entretanto, o *Bitcoin* possui limitações em termos de capacidades de programação para suportar transações complexas, o que o torna inadequado para a construção de aplicações distribuídas elaboradas.

A segunda geração de*blockchains*, exemplificada pelo *Ethereum*, surge para superar essas limitações e possibilitar a criação de aplicações distribuídas complexas além das criptomoedas. Nesse contexto, os contratos inteligentes, discutidos na seção subsequente, são destacados como o elemento central. O *blockchain Ethereum*, sendo o mais popular para o desenvolvimento de contratos inteligentes, é uma plataforma pública que incorpora uma linguagem Turing-completa,

permitindo a construção de qualquer contrato inteligente e aplicativo descentralizado (Alharby; Moorsel, 2017).

2.1.4 Identidade Digital Autossoberana Baseada em *Blockchain*

O Gerenciamento de Identidade (IdM) engloba os processos e políticas que regem o ciclo de vida dos atributos relacionados às identidades de um domínio específico. Atualmente, a maioria dos modelos de IdM é centralizada, onde uma entidade única exerce controle sobre todo o sistema. No entanto, as próprias identidades geradas podem ser federadas para além de uma única organização, como no caso de governos emitindo carteiras de identidades nacionais.

Em sistemas de identidade federada, os usuários têm a capacidade de utilizar informações de identidade diretamente nas mãos dos indivíduos. Isso inclui gerenciadores de senhas, como *1Password*, *Less-Pass*, entre outros, que de maneira segura gerenciam diversas credenciais em sites na internet (Nakamura *et al.*, 2019).

Apesar das diferentes abordagens, uma função crucial para o IdM é a vinculação segura de um identificador único, um valor distintivo que diferencia inequivocamente um usuário de outro dentro de um mesmo domínio. Além disso, há os atributos, também chamados de certificações ou declarações, que representam os direitos ou propriedades de um usuário, como nome, idade, classificação de crédito, entre outros.

Os primeiros esforços para adaptar o uso de Tecnologia de Livro-Razão Distribuído(DLT) visando estabelecer um mapeamento seguro e descentralizado de identificadores foram realizados no design do *Namecoin*, um *fork* distante do *Bitcoin*. O *Namecoin* proporciona um namespace legível, descentralizado e seguro para o domínio ".bit". Essa conquista desafiou a crença convencional de que um sistema de nomenclatura exibindo todas essas características não poderia ser concebido. O *Blockstack* expandiu esse esquema, criando uma infraestrutura de Infraestrutura de Chave Pública (PKI) descentralizada, registrando as ligações entre uma chave pública e um identificador legível.

Recentemente, surgiram diversos modelos de identidade descentralizada que vão além da nomenclatura e buscam oferecer um conjunto mais abrangente de funcionalidades IdM. No entanto, até o momento, não houve uma avaliação direta e abrangente dessas propostas.

Atualmente, existem duas principais categorias de propostas de IdM baseado em *blockchain*:

- ***Self-Sovereign Identity****(SSI)*: Refere-se a uma identidade que pertence e é controlada pelo seu proprietário, sem depender de qualquer autoridade administrativa externa e sem a possibilidade de remoção dessa identidade. Pode ser habilitado por um ecossistema de identidade descentralizado que facilita o registro e a troca de atributos de identidade, bem como a propagação da confiança entre as entidades participantes (Nakamura *et al.*, 2019).

- **Identidade Confiável Descentralizada:** Refere-se a uma identidade fornecida por um serviço centralizado que realiza a verificação de identidade dos usuários com base em credenciais confiáveis existentes (por exemplo, passaporte) e registra atestados de identidade em um DLT para validação posterior por terceiros (Nakamura *et al.*, 2019).

2.1.5 *Ethereum*

Criada em 2014 por Vitalik Buterin, *Ethereum* é uma plataforma de software descentralizada e de código aberto que possibilita a execução de programas conhecidos como contratos inteligentes (*smart contracts*). O objetivo do *Ethereum* é unir e aprimorar alguns conceitos de script, permitindo que os desenvolvedores criem aplicativos arbitrários baseados em consenso que incorporem a escalabilidade, padronização, funcionalidade completa, facilidade de desenvolvimento e interoperabilidade oferecidas por esses diferentes paradigmas simultaneamente.

O *Ethereum* alcaça esse propósito construindo essencialmente a melhor camada fundamental abstrata, um *blockchain* com uma linguagem de programação Turing-completa integrada. Isso possibilita que qualquer pessoa escreva contratos inteligentes e aplicativos descentralizados nos quais podem estabelecer suas próprias regras de propriedade, formatos e funções de transição de estado (Antonopoulos; Wood, 2018).

A unidade monetária do *Ethereum* é chamada de *ether*, identificada também como "ETH"ou com os símbolos Ξ (da letra grega “Xi” que se parece com um E maiúsculo estilizado) ou, menos frequentemente, ♢: para por exemplo, 1 ether ou 1 ETH ou Ξ1 ou ♢1 (Antonopoulos; Wood, 2018). O *Ether* é subdividido em unidades menores, até a menor unidade possível, denominada wei. Um ether equivale a 1 quintilhão de wei ($1 * 10^{18}$ ou 1.000.000.000.000.000.000). Você pode ouvir as pessoas se referirem à moeda como "*Ethereum*", mas isso é um erro comum. *Ethereum* é o sistema, e *ether* é a moeda.

2.1.6 Contratos Inteligentes *(Smart Contracts)*

Um contrato inteligente consiste em um código executável que opera no *blockchain* para facilitar, implementar e garantir a execução dos termos de um acordo. A principal finalidade de um contrato inteligente é realizar automaticamente os requisitos estipulados em um acordo assim que as condições previamente especificadas são atendidas. Dessa forma, os contratos inteligentes prometem taxas de transação mais baixas em comparação com sistemas convencionais, que dependem de terceiros confiáveis para fazer valer e executar os termos de um acordo. A concepção dos contratos inteligentes originou-se de Sazbo em 1994 (Alharby; Moorsel, 2017).

Os contratos inteligentes são imutáveis, ou seja, uma vez implantado seu código não pode ser alterado. Diferente dos *softwares* que utilizamos atualmente, a única forma de se alterar um contrato inteligente é implantando uma nova instância dele. Também operam em um contexto limitado, esses contratos tem acesso ao seu próprio estado, as transações que são realizadas

quando são utilizados e acesso a algumas informações de blocos recentes utilizados.

Os contratos inteligentes têm se mostrado uma ferramenta revolucionária no mundo atual, permitindo a automação e a segurança em diversas áreas. Por exemplo, no setor imobiliário, contratos inteligentes podem ser utilizados para facilitar a compra e venda de propriedades, onde o pagamento e a transferência de propriedade são automaticamente executados assim que as condições acordadas são atendidas, eliminando a necessidade de intermediários.

Na indústria financeira, esses contratos podem ser empregados em serviços de empréstimos, onde as condições de pagamento e juros são programadas diretamente no código, garantindo que as transações sejam realizadas de forma transparente e sem fraudes. Além disso, na cadeia de suprimentos, contratos inteligentes podem rastrear a origem e a movimentação de produtos, assegurando que todas as partes envolvidas cumpram suas obrigações, desde o produtor até o consumidor final.

2.1.7 *Solidity*

Solidity é uma linguagem de programação de alto nível e orientada a objetos utilizada para a implementação de contratos inteligentes, é uma linguagem que utiliza chaves delimitadoras, projetada para ser compatível com Máquina Virtual Ethereum (EVM). Teve influência de linguagens como C++, *Python* e *JavaScript*. É uma linguagem estaticamente tipada, ou seja, é necessário informar explicitamente o tipo de cada variável utilizada no sistema, suporta herança e tipos complexos definidos pelo usuário.

2.1.8 *Javascript*

A linguagem *JavaScript* foi criada pela Netscape Communications Corporation em 1995, posteriormente renomeada para *LiveScript*, e finalmente lançada como *JavaScript* no navegador Netscape. Seu propósito original era fornecer uma tecnologia de processamento cliente-side (lado do cliente), permitindo que os navegadores web executassem scripts diretamente no dispositivo do usuário, sem a necessidade de interação contínua com o servidor.

A principal função do *JavaScript* é permitir a criação de pequenos programas embutidos em páginas HTML, que podem executar uma série de tarefas diretamente no navegador do cliente. Essas tarefas incluem gerar números, processar dados, validar formulários, modificar e criar elementos HTML de forma dinâmica. Como resultado, *JavaScript* proporciona uma experiência mais fluida e interativa para os usuários, pois evita a necessidade de constantes trocas de dados com o servidor (Flanagan; Matilainen, 2007).

A linguagem *JavaScript*, além de sua relevância histórica e atual no desenvolvimento web, tornou-se essencial no desenvolvimento de contratos inteligentes, particularmente em plataformas baseadas em *blockchain* como o *Ethereum*. Uma das ferramentas mais proeminentes que utiliza *JavaScript* para facilitar esse processo é o *HardHat*. O *Hardhat* integra-se perfei-

tamente com *JavaScript* (ou *TypeScript*), fornecendo uma API que permite a escrita de testes automatizados. Esses testes simulam interações com os contratos inteligentes, verificando sua funcionalidade e segurança em diversas condições. Testar contratos inteligentes é essencial devido à sua imutabilidade após a implantação no *blockchain*.

2.1.9 *HardHat*

HardHad é um ambiente de desenvolvimento de software *Ethereum* que inclui vários componentes, como ferramentas de compilação, depuração e implantação de contratos inteligentes. O ambiente de desenvolvimento de contratos inteligentes do Hardhat oferece aos programadores recursos para gerenciar seu fluxo de trabalho. Além disso, o Hardhat otimiza o processo de desenvolvimento automatizando certas etapas e introduzindo novas funcionalidades produtivas.

O *Hardhat* é fornecido com uma rede *Ethereum* local pré-configurada, projetada especificamente para propósitos de desenvolvimento. Também disponibiliza suporte abrangente para mensagens de erro, proporcionando aos programadores as ferramentas necessárias para identificar a instância exata. Com essas funcionalidades, os desenvolvedores estão capacitados para detectar e corrigir eficientemente as falhas da aplicação. Ele também pode fornecer a solução necessária para resolver os problemas subjacentes à falha da aplicação(Kuonen, 2023).

2.2 Trabalhos Correlatos

A gestão de identidade é um tema que vem se tornando bem relevante no mundo atual interconectado. Existem algumas soluções desenvolvidas para contornar o problema de vazamento de dados. Esta seção apresenta algumas ferramentas que tem o propósito de evitar tais diversidades.

2.2.1 Hyperledger Besu

O Hyperledger Besu é uma solução de código aberto compatível com a *Ethereum*, sua compatibilidade com esse ecossistema permite que ele execute contratos inteligentes e aplicativos descentralizados. Ele também possibilita um elevado nível de privacidade por meio de diferentes protocolos de segurança (Pinto *et al.*, 2023). O Besu é o mais adequado para o desenvolvimento de aplicações financeiras que exigem segurança e alto desempenho no processamento de transações privadas, uma vez que é escalável, confiável e oferece um alto nível de privacidade (Praitheeshan; Pan; Doss, 2021).

2.2.2 Hyperledger Fabric

Hyperledger Fabric é uma rede de *blockchain* de código aberto, desenvolvida pelo projeto Hyperledger, é o projeto mais maduro da iniciativa Hyperledger utilizado em soluções de nível empresarial, incluindo gestão descentralizada de DID (Pinto *et al.*, 2023). A arquitetura modular

do Fabric permite a utilização de diversos tipos de protocolos de consenso, diferentes linguagens de programação para a criação de contratos inteligentes e diferentes bancos de dados para armazenamento das correntes de bloco e seu estado atual (Rebello *et al.*, 2019).

O Fabric permite a criação de redes que abrigam diversas cadeias de blocos isoladas, independentes entre si e com configurações, contratos inteligentes e participantes distintos. Os participantes de uma rede baseada em blocos Fabric desempenham diversos papéis, com diferentes permissões. Este software oferece uma flexibilidade para o estabelecimento de políticas específicas e adaptadas para a validação das transações, ampliando assim o controle sobre a gestão de identidades descentralizadas (Rebello *et al.*, 2019).

2.2.3 Hyperledger Indy

o Hyperledger Indy também é uma solução de código aberto para o gerenciamento de Identidades Auto Soberanas. Ele é projetado para permitir que os usuários tenham total controle sobre suas identidades e garante a privacidade. Com essa ferramenta, é possível criar credenciais verificáveis e selecionar quais informações podem ser compartilhadas com entidades confiáveis (Pinto *et al.*, 2023).

A singularidade das identidades no Indy é ressaltada pelos DIDs, que são globalmente únicos e resolvíveis sem a dependência de qualquer autoridade centralizada. Essas características proporcionam ao Indy a capacidade de estabelecer identidades em contratos inteligentes de outros sistemas, permitindo assim uma integração eficiente (Priya; Ponnavaikko; Aantonny, 2020). Além disso, o software implementa mecanismos de preservação de privacidade, como Provas de Conhecimento Zero e Assinaturas Digitais. Um exemplo prático de uma solução baseada em Indy é a *blockchain Sovrin*, que oferece uma plataforma DID de propósito geral (Baniata *et al.*, 2022).

2.2.4 uPort

O uPort é uma plataforma, também de código aberto, que fornece uma identidade descentralizada para uma identidade Auto Soberana (Naik; Jenkins, 2020). É baseado no *Ethereum* e utiliza os *smart contracts* criados na *blockchain*. Ao utilizar esta estrutura, os usuários podem publicar sua identidade, fazer transferências dos dados de sua credencial e controlar as chaves e dados com segurança.

As identidades criadas utilizando o uPort, são vinculadas de forma criptográfica em sistemas de armazenamento. Cada identidade é capaz de guardar um hash, onde as informações são armazenadas de forma segura. As identidades são capazes de atualizar seu próprio arquivo, como por exemplo adicionar uma foto de perfil, ou também conceder permissões de leituras temporárias (Lundkvist *et al.*, 2017).

Este software utiliza um identificador uPort, onde uma *string* hexadecimal atua como

identificador, esse identificador é definido como o endereço de um contrato inteligente *Ethereum*. Este contrato pode retransmitir transações, a partir desse modo que a identidade interage com os demais contratos inteligentes no *blockchain* (Lundkvist *et al.*, 2017).

3 Desenvolvimento

Neste capítulo são apresentados as ferramentas e métodos utilizados para o desenvolvimento desse trabalho. Nas seções a seguir serão apresentadas as definições e detalhes das principais estratégias e metodologias utilizadas na abordagem proposta.

3.1 Rede de *blockchain Ethereum*

A *Ethereum* foi a *blockchain* escolhida para a implementação deste trabalho, uma vez que, ela oferece um suporte nativo para implementações de contratos inteligentes tornando possível uma implementação eficiente da lógica necessária para o sistema de gestão de identidade descentralizada, a linguagem *Solidity*, foi desenvolvida para a *Ethereum* o que proporciona uma abordagem simples e robusta para a realização do contrato inteligente. A Máquina Virtual *Ethereum* desempenha um papel importante na execução desse contrato, sua arquitetura facilita a execução segura assim garantindo a confiabilidade do sistema.

3.2 Remix IDE

O RemixIDE foi o ambiente de desenvolvimento escolhido para a implementação do contrato inteligente deste trabalho, já que, é um ambiente que se encontra no próprio domínio do *Ethereum* e é utilizado direto no navegador, que permite além do desenvolvimento do código do contrato, realizar testes e integração com o *blockchain*.

3.3 Especificação de Requisitos

Nesta seção é apresentado os principais requisitos para a implementação do contrato inteligente para Gestão de Identidades Descentralizada. Após uma análise, os requisitos a seguir indicam os principais características para garantir a maior eficácia.

3.3.1 Requisitos Funcionais

Os requisitos funcionais representam as principais funcionalidades que devem ser inclusas no contrato inteligente. A tabela a seguir são especificados esses requisitos.

Requisito Funcional	Descrição
Registro de Identidade	Este requisito tem como principal função estabelecer uma forma segura e eficiente para que os usuário registrem suas identidades digitais no blockchain
Autenticação Descentralizada	Este exigência estabelece a criação de um sistema sólido para a comprovação da identidade, onde é atribuído a cada usuário um par de chaves criptográficas, uma privada e uma pública.
Controle de Acesso	Esta condição procura estabelecer um sistema flexível que permita aos usuários gerenciarem suas próprias permissões.
Revogação de Acesso	Este requisito permite ao usuário um controle imediato sobre as permissões concedidas, permitindo uma restrição a determinados dados. Conferindo ao usuário o poder de gerenciar quem tem acesso a suas informações.
Armazenamento Descentralizado de Dados	Este requisito refere-se a necessidade de armazenar as informações de forma distribuída e segura na *blockchain*, com um modelo descentralizado existe a possibilidade de eliminar pontos de falhas, tornando o sistema mais robusto. Cada usuário mantém o controle exclusivo de suas informações, que são guardados de forma imutável na *blockchain*.

Tabela 3.1 – Tabela de Requisitos Funcionais

3.3.2 Requisitos Não Funcionais

Os requisitos não funcionais descrevem as restrições do contrato inteligente. A tabela a seguir são especificados esses requisitos.

Requisito Não Funcional	Descrição
Segurança	Este requisito tem como principal função estabelecer uma forma segura e eficiente para que os usuário registrem e atualizem suas identidades digitais no blockchain.
Eficiência e Escalabilidade	Este requisito evidência a necessidade de garantir tanto a eficiência operacional do contrato, como também a capacidade de expansão para a acomodação de um aumento na quantidade de identidades a serem gerenciadas. A escalabilidade é abordada por meio de estratégias que admitam a adição de novos usuários sem comprometer a performance. Para a eficiência, funções otimizadas de processamento e transações serão implementadas para garantir tempos de resposta efetivos.
Interoperabilidade	Este requisito promove a integração do sistema com outras plataformas e redes descentralizadas, possibilitando uma troca eficiente de informações entre diferentes ecossistemas. Ao garantir a interoperabilidade, facilitando a colaboração e a utilização das identidades em diversas aplicações.
Privacidade	Este requisito é relacionado com à proteção dos dados sensíveis dos usuários, garantindo que as informações pessoais sejam tratadas com confidencialidade

Tabela 3.2 – Tabela de Requisitos Não Funcionais

3.4 Análise dos Requisitos

Os requisitos levantados na seção anterior, indicam uma abordagem abrangente e estratégica para construir um sistema sólido e eficiente. O requisito de Registro de Identidade da ênfase na segurança e eficiência para a realização de um registro de uma identidade de maneira confiável no *blockchain*. O desenvolvimento desse requisito estabelece uma infraestrutura confiável para transações digitais e promove a autenticidade das identidades.

A condição de Autenticação Descentralizada reforça a segurança ao atribuir ao usuário pares de chaves criptográficas, realçando a importância de um sistema de comprovação de identidade, criando uma camada de proteção para as informações registradas, garantindo a singularidade e a confidencialidade, fundamentais para a integridade do sistema.

O requisito de Controle de Acesso levanta a preocupação com a autonomia do usuário, desse modo, permitindo o gerenciamento de suas próprias permissões de maneira flexível, re-

forçando a privacidade. A exigência de Revogação de Acesso confere ao usuário um controle imediato e dinâmico sobre suas permissões, proporcionando uma camada adicional de segurança. A capacidade de restringir o acesso a determinados dados promove a segurança e coloca o usuário como controlador de quem pode acessar suas informações.

O requisito de Armazenamento Descentralizado de Dados destaca a importância de uma abordagem distribuída para garantir a segurança do sistema, esta condição reforça a confiabilidade do sistema, ao mesmo tempo que entrega para o usuário o controle de suas informações, guardadas de forma imutável na *blockchain*.

Os requisitos não funcionais adicionam uma camada de abrangência e eficácia ao sistema. O requisito de Segurança é reforçado, não apenas no registro, mas também na atualização contínua das identidades digitais. Eficiência e Escalabilidade são destaques essenciais para garantir tempos de resposta efetivos e a capacidade de expansão do sistema, atendendo a um aumento na quantidade de identidades a serem gerenciadas.

A Interoperabilidade tem como estratégia principal promover a troca eficiente de informações com outras plataformas descentralizadas, estimulando a colaboração e ampliando as possibilidades de uso das identidades digitais. Por fim, o requisito de Privacidade reforça o compromisso com a proteção dos dados sensíveis dos usuários, garantindo que as informações pessoais sejam manuseadas com confidencialidade. A integração desses requisitos funcionais e não funcionais forma uma base sólida e abrangente para o desenvolvimento de um sistema de gestão de identidade descentralizada inovador e confiável.

3.5 Elaboração do Contrato Inteligente

A construção do contrato inteligente tem a intenção de oferecer um ambiente seguro e descentralizado para o gerenciamento de identidades na *blockchain*. Através dele, é possível criar, atualizar e consultar os dados registrados de forma transparente, mantendo assim, a privacidade e a integridade das informações. Nesta seção, apresento o objeto Pessoa, que recebe as informações do usuário e as funções que representam os requisitos funcionais apresentados nesse trabalho.

3.5.1 Estrutura do objeto Pessoa

A implementação desse contrato inteligente se inicia pela definição de um objeto *Person*. Esse objeto é essencial, já que, ele estrutura as informações pessoais de cada usuário que é registrado no sistema. Em um sistema de gerenciamento de identidades descentralizado, é fundamental manter diversos dados identificatórios que estão integrados a essa estrutura.

A seguir uma explicação dos campos selecionados para a estruturação do objeto *Person*:

- `name`: Este campo armazena o nome do usuário, é um dado principal para a identificação básica em sistemas de autenticação e verificação. O uso de uma string permite a flexibilidade na quantidade de caracteres que podem ser utilizados, adequando-se aos diferentes comprimentos de nomes.
- `dateOfBirth`: A data de nascimento da pessoa, também armazenada como uma string, porém existe uma função para validar sua estrutura. Este dado é importante para verificações relacionadas a contratos com restrições de idade.
- `idDocument`: O documento de identificação é o campo mais importante, uma vez que, cada usuário tem um número único de identificação, que pode ser CPF, passaporte, ou qualquer outro documento que sirva para a verificação legal da identidade do indivíduo. Armazenar esse dado facilita a integração com sistemas que exigem autenticação com documentos únicos.
- `addressLocation`: Remete ao endereço físico da pessoa. Em sistemas que necessitam de verificação de endereço ou envio de correspondência, este campo se torna bem relevante. Além disso, é um dado importante para garantir que o sistema possa vincular identidades digitais a locais físicos.
- `phone`: O telefone, um dado que facilita a comunicação com o usuário. Em muitos sistemas, a verificação via SMS é uma medida adicional de segurança, o que torna esse campo bastante utilizado em sistemas que exigem múltiplos fatores de autenticação.
- `nationality`: Este campo armazena a nacionalidade da pessoa, o que pode ser relevante para sistemas que exigem restrições geográficas ou para controle de origem de cada usuário.

- `naturalness`: Refere-se à naturalidade, ou seja, o local de nascimento do indivíduo. Este campo pode ser utilizado para verificar origens regionais.
- `ethnicity`: O campo de etnia é incluído para sistemas que necessitam dessa informação para fins estatísticos ou de inclusão social.
- `exists`: Este é um campo booleano que verifica se o registro da pessoa já existe no sistema. Ele evita que duplicidades ocorram, garantindo que um usuário não seja cadastrado mais de uma vez no sistema. Essa verificação é essencial para assegurar a integridade do banco de dados de identidades.

A seleção desses campos têm como objetivo garantir que o contrato armazene informações relevantes para a identificação de um usuário de uma forma segura e eficiente. A utilização dessa estrutura permite que o sistema utilize as funções de criação e de atualização de dados de forma consistente, utilizando o controle de acesso necessário. Cada endereço da *blockchain Ethereum* está associado a um objeto *Person*, possibilitando de forma eficiente, o mapeamento de cada identidade digital cadastrada no sistema.

3.5.2 Funções de Criação e Atualização de Identidades

Nesta seção será abordado as explicações de cada uma das funções, a função `createPerson` que permite a criação de uma identidade e a função `updatePerson`, que tem como objetivo atualizar os dados.

3.5.2.1 Função de Criar Identidade

A função de criar identidade tem a responsabilidade de registrar os dados do usuário para a criação de uma nova identidade. Tem como parâmetros as informações pessoais básicas, como nome, data de nascimento, um documento de identificação, um endereço físico, um número de telefone (este pode ser tanto um número de telefone fixo, quanto um número de telefone celular), a nacionalidade e a naturalidade do usuário que está fazendo o registro e sua etnia.

Listing 3.1 – Função de criar identidade

```
1    // Funcao para criar uma identidade de pessoa
2    function createPerson(
3        string memory _name,
4        string memory _dateOfBirth,
5        string memory _idDocument,
6        string memory _addressLocation,
7        string memory _phone,
8        string memory _nationality,
9        string memory _naturalness,
```

```
10          string memory _ethnicity
11      ) public {
12          require(!persons[msg.sender].exists, "Person already exists
                ");
13          require(isValidDate(_dateOfBirth), "Invalid date format,
                use DD/MM/YYYY");
14
15          persons[msg.sender] = Person({
16              name: _name,
17              dateOfBirth: _dateOfBirth,
18              idDocument: _idDocument,
19              addressLocation: _addressLocation,
20              phone: _phone,
21              nationality: _nationality,
22              naturalness: _naturalness,
23              ethnicity: _ethnicity,
24              exists: true
25          });
26
27          emit PersonCreated(msg.sender, _name);
28      }
```

Dentro dessa função se encontra também duas verificações muito importantes, a primeira que garante de que cada usuário possa se cadastrar somente uma única vez, não permitindo a criação de mais de uma identidade. A verificação acontece por meio da condição `require(!persons[msg.sender].exists, "Person already exists")`, onde ela retorna do sistema se o usuário já está cadastrado ou não, caso a condição dentro `require()` for verdadeira o sistema retorna que o usuário não foi cadastrado e permite continuar o processo, caso contrário, o sistema interrompe o processo. A segunda verificação é o formato da data de nascimento que utiliza a função `require(isValidDate(_dateOfBirth), "Invalid date ormat, use DD/MM/YYYY")`, ela verifica se a `string dateOfBirth` contém a quantidade de números permitidos, se contém as barras necessárias e valida também o ano de nascimento do usuário, não permitindo cadastrar anos abaixo de 1500 e acima do ano atual.

3.5.2.2 Função de Atualizar Identidade

A função `updatePerson` permite que o usuário possa atualizar as informações de uma identidade existente no sistema. Tem basicamente o mesmo fluxo da função de criação, porém essa função só pode ser utilizada por usuários autorizados, essa verificação é feita pelo `onlyAuthorized`, somente após a verificação e das atualizações aconteceram o evento de atualizar é realizado.

Listing 3.2 – Função de atualizar identidade

```
    function updatePerson(
        string memory _name,
        string memory _dateOfBirth,
        string memory _idDocument,
        string memory _addressLocation,
        string memory _phone,
        string memory _nationality,
        string memory _naturalness,
        string memory _ethnicity
    ) public onlyAuthorized {
        require(persons[msg.sender].exists, "Person does not exist");
        require(isValidDate(_dateOfBirth), "Invalid date format, use DD/MM/YYYY");

        persons[msg.sender] = Person({
            name: _name,
            dateOfBirth: _dateOfBirth,
            idDocument: _idDocument,
            addressLocation: _addressLocation,
            phone: _phone,
            nationality: _nationality,
            naturalness: _naturalness,
            ethnicity: _ethnicity,
            exists: true
        });

        emit PersonUpdated(msg.sender, _name);
    }
```

3.5.3 Funções de Acesso e Controle de Permissões

Para controle de acesso das informações foram implementadas três funções, a primeira delas é a `grantAccess` que é usada para conceder acessos aos dados, a segunda é a `revokeAccess` que permite ao dono das informações a revogar os acessos de outros usuários aos seus próprios dados e a última é a `isAuthorized` que verifica se o usuário possui ou não autorização. A seguir será apresentada cada uma delas.

3.5.3.1 Função grantAccess

A função `grantAccess()`, como dito anteriormente foi implementada para que o usuário garanta acesso de terceiros a suas informações. Para esse acesso ser feito de forma segura, a função tem como parâmetro um argunto do tipo `address`, esse parâmetro identifica o usuário que deseja acessar os dados por meio da *blockchain Ethereum*. Além disso a função possui o modificador `onlyAdmin`, possibilitando a execução apenas pelo usuário administrador(o dono das informações).

Listing 3.3 – Função grantAccess

```
1    // Funcao de autenticacao para autorizar usuarios
2    function grantAccess(address _user) public onlyAdmin {
3        require(!authorized[_user], "User already authorized");
4        authorized[_user] = true;
5        emit AccessGranted(_user);
6    }
```

Quando a função é chamada ela executa a instrução `require()` para verificar se o usuário que está tentando acessar as informações foi autorizado ou não. Caso o acesso tenha sido autorizado, ou seja, se o endereço do terceiro venha marcado como *true* no mapameanto `authorized`, a execução da função é interrompida e a mensagem `"User already authorized"` é enviada. Caso contrário, o contrato modifica o mapeamento `authorized` marcando como *true*, indicando que agora o usuário possui as permissões necessárias para o acesso. Após essa atualização, a função emite o evento `AccessGranted`, que registra a concessão de acesso no log da blockchain, permitindo a auditoria e o rastreio das ações.

3.5.3.2 Função revokeAccess

A função `revokeAccess` remove o acesso previamente concedido a um determinado usuário pertencente ao contrato inteligente. Recebe como parâmetro o endereço do usuário, assim podendo identifica-lo na rede *blockchain* e revogando o acesso as informações e como dito na função anterior, também é protegida pelo modificador `onlyAdmin`, reforçando que apenas o administrador possa executar essa função.

Listing 3.4 – Função revokeAccess

```
1    function revokeAccess(address _user) public onlyAdmin {
2        require(authorized[_user], "User not authorized");
3        authorized[_user] = false;
4        emit AccessRevoked(_user);
5    }
```

A estrutura é parecida com a função `grantAccess`, por meio da instrução `require()`, verifica se o usuário possui uma autorização, ou seja, se o endereço do usuário está com o mapeamento

`authorized` marcado como *true*. Se o usuário não estiver com essa marcação, significa que o usuário não tem autorização, a execução é interrompida e a mensagem de erro `"User not authorized"` é lançada. Caso contrário, é modificado o mapeamento `authorized`, o definindo como `false` e assim removendo o acesso do usuário. Após essa atualização a função registra o evento `AccessRevoked`, registrando a revogação de acesso no log da *blockchain*.

3.5.3.3 Função isAuthorized

Essa função é utilizada para verificar se um determinado usuário possui ou não as permissões necessárias de acesso. É uma função simples, mas bastante eficiente para uma verificação rápida sobre o status de autorização de um determinado usuário.

Listing 3.5 – Função isAuthorized

```
1    function isAuthorized(address _user) public view returns (bool)
         {
2        return authorized[_user];
3    }
```

4 Resultados

Nesta seção será apresentado os resultados obtidos na construção do contrato inteligente de acordo com as etapas apresentadas na seção anterior. Foi implementado um conjunto de testes automatizados com o objetivo de avaliar a segurança e a funcionalidade do contrato inteligente para gestão de identidade. Para essa finalidade foram utilizadas a tecnologia *HardHat* e a linguagem *Javascript*. A primeira teve seu papel como o ambiente para o desenvolvimento, compilação e execução dos testes, oferecendo uma ótima opção para simulações na *blockchain Ethereum*. A linguagem *Javascript* foi utilizada para a implementação dos teste realizados.

Os testes realizados demostram a capacidade de criar identidades, fazer o controle de acesso as informações do usuário e permitem gerenciar as permissões de terceiros para visualização ou alteração de dados, foram realizados 6 testes e todos foram concluídos com sucesso.

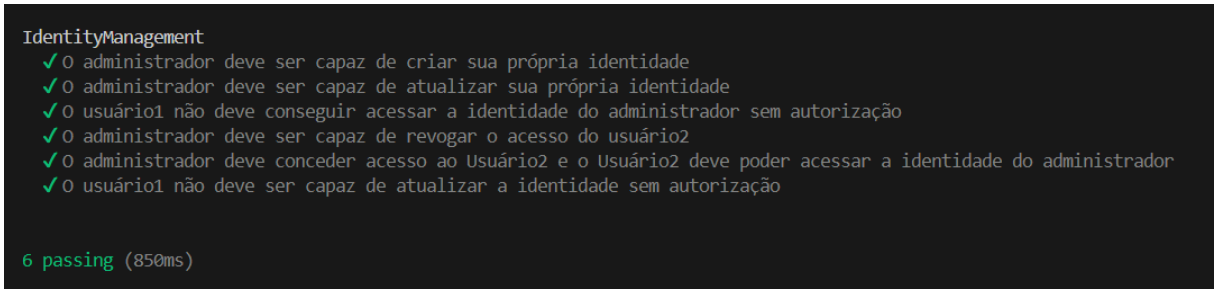

Figura 4.1 – Testes realizados

Antes da execução de todos os teste, primeiro é necessário fazer o *deploy* do contrato inteligente no ambiente, realizar a criação das variáveis que será atribuída a fábrica do contrato, a instância do contrato e os usuários que tem acesso ao contrato. O seguinte trecho de código faz esse processo.

Listing 4.1 – Declaração e deploy do contrato inteligente

```
describe("IdentityManagement", function () {
let IdentityManagement, identityContract, admin, user1, user2;
beforeEach(async function () {

    IdentityManagement = await ethers.getContractFactory("
       IdentityManagement");
    [admin, user1, user2] = await ethers.getSigners();

    identityContract = await IdentityManagement.deploy();
});
...
}
```

4.0.1 Teste de criação de identidade

O primeiro teste a ser realizado foi o de criação de identidade, onde o usuário administrador cria sua própria identidade utilizando a função `createPerson`, onde são passados parâmetros como nome, data de nascimento, documento de identidade, endereço, entre outros. Esses dados são inseridos na *blockchain*, e o teste é projetado para verificar se, após essa operação, a identidade recém-criada pode ser acessada corretamente. Utiliza-se a função `getPerson` para buscar os dados armazenados e compará-los com os fornecidos no momento da criação. No teste, por exemplo, o administrador cria sua própria identidade e os dados são comparados para garantir que foram registrados corretamente, assegurando a integridade das informações no contrato.

Listing 4.2 – Teste de criação de identidade

```
it("O administrador deve ser capaz de criar sua propria
   identidade", async function () {
  await identityContract.connect(admin).createPerson(
    "Admin User",
    "01/01/1990",
    "123456",
    "Admin Address",
    "123-456-7890",
    "Nationality",
    "Naturalness",
    "Ethnicity"
  );

  const person = await identityContract.getPerson(admin.address);
  expect(person.name).to.equal("Admin User");
```

```
15  });
```

4.0.2 Atualização de Identidade pelo Administrador

Neste teste, verifica-se a capacidade do administrador de atualizar os dados de uma identidade já existente. O processo envolve a chamada da função `updatePerson`, na qual o administrador fornece os novos dados da identidade. A ideia é garantir que, após a execução da função de atualização, os dados antigos sejam substituídos corretamente pelos novos valores. O teste recupera os dados atualizados e compara com os valores que foram fornecidos, garantindo que a modificação ocorreu conforme esperado. A segurança é um fator bem importante aqui, pois o teste também assegura que apenas o administrador, ou uma entidade autorizada, tenha a permissão para realizar essa atualização.

Listing 4.3 – Teste de atualização de identidade

```
1   it("O administrador deve ser capaz de atualizar sua propria
        identidade", async function () {
2
3     await identityContract.connect(admin).createPerson(
4       "Admin User",
5       "01/01/1990",
6       "123456",
7       "Admin Address",
8       "123-456-7890",
9       "Nationality",
10      "Naturalness",
11      "Ethnicity"
12    );
13
14    await identityContract.connect(admin).updatePerson(
15      "Updated Admin User",
16      "01/01/1985",
17      "654321",
18      "Updated Admin Address",
19      "321-654-0987",
20      "Updated Nationality",
21      "Updated Naturalness",
22      "Updated Ethnicity"
23    );
24
25    const updatedPerson = await identityContract.getPerson(admin.
          address);
```

```
26
27      expect(updatedPerson.name).to.equal("Updated Admin User");
28      expect(updatedPerson.dateOfBirth).to.equal("01/01/1985");
29      expect(updatedPerson.idDocument).to.equal("654321");
30      expect(updatedPerson.addressLocation).to.equal("Updated Admin
            Address");
31      expect(updatedPerson.phone).to.equal("321-654-0987");
32      expect(updatedPerson.nationality).to.equal("Updated Nationality
            ");
33      expect(updatedPerson.naturalness).to.equal("Updated Naturalness
            ");
34      expect(updatedPerson.ethnicity).to.equal("Updated Ethnicity");
35    });
```

4.0.3 Proibição de Acesso à Identidade do Administrador por Outros Usuários

Este teste é essencial para garantir que usuários não autorizados não consigam acessar informações sensíveis, como a identidade do administrador. O código tenta realizar uma chamada à função `getPerson` usando a conta de um usuário não autorizado, no caso, o `user1`. O teste espera que essa tentativa seja bloqueada pelo sistema, resultando em um erro específico, que informa que o acesso foi negado com a mensagem `"Access denied: unauthorized user"`. Essa verificação demonstra que o controle de acesso implementado no contrato inteligente está funcionando adequadamente, protegendo os dados pessoais de acessos indevidos.

Listing 4.4 – Teste de proibição de acesso

```
1     it("O usuario1 nao deve conseguir acessar a identidade do
          administrador sem autorizacao", async function () {
2
3     await expect(identityContract.connect(user1).getPerson(admin.
          address))
4           .to.be.revertedWith("Access denied: unauthorized user");
5     });
```

4.0.4 Revogação de Acesso

O teste de revogação de acesso garante que o administrador pode revogar permissões de acesso previamente concedidas a outros usuários. O fluxo do código envolve o administrador concedendo inicialmente o acesso de visualização de identidade para o `user2` utilizando a função `grantAccess`. Em seguida, o acesso é revogado com a função `revokeAccess`. O teste deve

garantir que, após essa revogação, o `user2` não consiga mais acessar os dados da identidade do administrador, e qualquer tentativa de fazê-lo deve resultar em um erro de acesso negado. Este teste confirma que o sistema permite ao administrador controlar dinamicamente as permissões de acesso, garantindo que elas sejam eficazmente removidas quando necessário.

Listing 4.5 – Teste para revogar acesso

```
1   it("O administrador deve ser capaz de revogar o acesso do
        usuario2", async function () {
2
3       await identityContract.connect(admin).createPerson(
4         "Admin User",
5         "01/01/1990",
6         "123456",
7         "Admin Address",
8         "123-456-7890",
9         "Nationality",
10        "Naturalness",
11        "Ethnicity"
12      );
13
14      await identityContract.connect(admin).grantAccess(user2.
            address);
15
16      await expect(identityContract.connect(user2).getPerson(
            admin.address))
17        .to.not.be.reverted;
18
19      await identityContract.connect(admin).revokeAccess(user2.
            address);
20
21      await expect(identityContract.connect(user2).getPerson(
            admin.address))
22        .to.be.revertedWith("Access denied: unauthorized user");
23    });
```

4.0.5 Concessão de Acesso a Outro Usuário

Neste teste, é verificado se o administrador pode conceder acesso à sua identidade para outro usuário, especificamente o `user2`. A função `grantAccess` é utilizada para adicionar o `user2` à lista de usuários autorizados a acessar a identidade do administrador. Após a concessão do acesso, o teste verifica se o `user2` é capaz de acessar a identidade do administrador, garantindo que

a operação é bem-sucedida. O sucesso deste teste assegura que o sistema permite a colaboração entre usuários, mas de maneira controlada, onde o administrador tem total controle sobre quem pode acessar suas informações pessoais.

Listing 4.6 – Teste para conceder acesso

```
1  it("O administrador deve conceder acesso ao Usuario2 e o Usuario2
       deve poder acessar a identidade do administrador", async
     function () {
2
3    await identityContract.connect(admin).createPerson(
4      "Admin User",
5      "01/01/1990",
6      "123456",
7      "Admin Address",
8      "123-456-7890",
9      "Nationality",
10     "Naturalness",
11     "Ethnicity"
12   );
13
14   await identityContract.connect(admin).grantAccess(user2.address
       );
15
16   await expect(identityContract.connect(user2).getPerson(admin.
       address))
17     .to.not.be.reverted;
18 });
```

4.0.6 Proibição de Atualização de Identidade por Usuários Não Autorizados

Este teste tem como foco assegurar que apenas usuários autorizados possam atualizar as informações de uma identidade. O código tenta realizar uma chamada à função `updatePerson` usando a conta de um usuário não autorizado, no caso, o `user1`. O teste verifica se essa chamada resulta em um erro, indicando que o usuário não tem permissão para realizar a atualização. Isso garante que as informações armazenadas no contrato só podem ser modificadas por aqueles que possuem autorização explícita, protegendo assim a integridade dos dados e evitando alterações não autorizadas.

Listing 4.7 – Teste onde nao e possivel alterar as informacoes

```
1  it("O usuario1 nao deve ser capaz de atualizar a identidade sem
       autorizacao", async function () {
2
3    await expect(identityContract.connect(user1).updatePerson(
4      "User1 Name",
5      "02/02/1995",
6      "654321",
7      "User1 Address",
8      "987-654-3210",
9      "Other Nationality",
10     "Other Naturalness",
11     "Other Ethnicity"
12   )).to.be.revertedWith("Access denied: unauthorized user");
13 });
```

5 Considerações Finais

5.1 Conclusão

Este trabalho teve como principal objetivo a implementação de um contrato inteligente utilizando a linguagem *Solidity* para a gestão de identidade descentralizada (DID), baseado na plataforma *Ethereum*. A partir das pesquisas realizadas, foram levantados os requisitos funcionais e não funcionais, finalizando com a criação de um sistema que oferece segurança, flexibilidade e controle aos usuários sobre suas identidades digitais.

A solução apresentada demonstrou eficiência na criação, atualização e gerenciamento de identidades, utilizando um sistema baseado em *blockchain*, onde cada usuário é responsável por suas próprias informações, sem depender de uma autoridade centralizada. Através dos testes realizados, foi possível verificar que o sistema atende às expectativas em termos de segurança e privacidade, garantindo que apenas usuários autorizados possam acessar ou modificar dados pessoais.

Os resultados obtidos mostram que o contrato inteligente desenvolvido é uma solução bem eficiente para a gestão de identidades descentralizadas, tanto em cenários pessoais quanto corporativos. A habilidade dos usuários em gerenciar o acesso, juntamente com a possibilidade de revogar permissões de forma eficaz, aumenta a possível utilização da plataforma em contextos que demandam um alto nível de segurança e privacidade.

5.2 Trabalhos Futuros

Para a ampliação da solução proposta, consideramos a inclusão de uma integração com uma aplicação front-end, proporcionando aos usuários uma experiência mais intuitiva no registro e gerenciamento de seus dados. Essa implementação não apenas aumentaria a visibilidade da solução, tornando-a acessível a um público mais amplo, mas também melhoraria a usabilidade para usuários comuns.

Outra direção para trabalhos futuros seria a adaptação do contrato inteligente para atender às necessidades de usuários comerciais, como empresas. A flexibilidade intrínseca ao sistema permitiria ajustes nas funções implementadas, facilitando, por exemplo, o cadastro eficiente de novos usuários no contexto corporativo. Essa expansão potencial abriria caminho para a utilização da solução em ambientes empresariais, explorando ainda mais os benefícios da gestão descentralizada de identidade.

Referências

ALHARBY, M.; MOORSEL, A. V. Blockchain-based smart contracts: A systematic mapping study. **arXiv preprint arXiv:1710.06372**, 2017.

ANTONOPOULOS, A. M.; WOOD, G. **Mastering ethereum: building smart contracts and dapps**. [S.l.]: O'reilly Media, 2018.

BANIATA, H. *et al.* Latency assessment of blockchain-based ssi applications utilizing hyperledger indy. In: **CLOSER**. [S.l.: s.n.], 2022. p. 264–271.

FLANAGAN, D.; MATILAINEN, P. **JavaScript**. [S.l.]: Anaya Multimedia, 2007.

KUONEN, D. The process of creating, testing, and deploying smart contracts on the ethereum blockchain using solidity. **Haaga-Helia University of Applied Sciences**, 2023.

LUNDKVIST, C. *et al.* Uport: A platform for self-sovereign identity. **URL: https://whitepaper. uport. me/uPort_ whitepaper_DRAFT20170221. pdf**, 2017.

NAIK, N.; JENKINS, P. uport open-source identity management system: An assessment of self-sovereign identity and user-centric data platform built on blockchain. In: IEEE. **2020 IEEE International Symposium on Systems Engineering (ISSE)**. [S.l.], 2020. p. 1–7.

NAKAMURA, E. T. *et al.* Identidade digital descentralizada: conceitos, aplicações, iniciativas, plataforma de desenvolvimento e implementação de caso de uso. **Sociedade Brasileira de Computação**, 2019.

PINTO, M. *et al.* Identidade descentralizada e blockchain: Um estudo exploratório sobre as oportunidades e desafios das soluções existentes. In: SBC. **Anais do I Colóquio em Blockchain e Web Descentralizada**. [S.l.], 2023. p. 43–48.

PRAITHEESHAN, P.; PAN, L.; DOSS, R. Private and trustworthy distributed lending model using hyperledger besu. **SN Computer Science**, Springer, v. 2, p. 1–19, 2021.

PRIYA, N.; PONNAVAIKKO, M.; AANTONNY, R. An efficient system framework for managing identity in educational system based on blockchain technology. In: IEEE. **2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)**. [S.l.], 2020. p. 1–5.

REBELLO, G. *et al.* Correntes de blocos: Algoritmos de consenso e implementação na plataforma hyperledger fabric. **Sociedade Brasileira de Computação**, 2019.

SIQUEIRA, A.; CONCEIÇÃO, A. F. da; ROCHA, V. Blockchain e identidades digitais descentralizadas: Fundamentos e oportunidades. **Fundamentos e Tendências em Inovação Tecnológica: Volume 2**, 2021.

Apêndices

APÊNDICE A – Contrato Inteligente em Solidity

Listing A.1 – Contrato inteligente em Solidity

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract IdentityManagement {
    struct Person {
        string name;
        string dateOfBirth;
        string idDocument;
        string addressLocation;
        string phone;
        string nationality;
        string naturalness; // Naturalidade
        string ethnicity;
        bool exists;
    }

    mapping(address => Person) private persons;
    mapping(address => bool) private authorized; // Controle de
       acesso

    address public admin;

    event AccessGranted(address indexed user);
    event AccessRevoked(address indexed user);
    event PersonCreated(address indexed user, string name);
    event PersonUpdated(address indexed user, string name);

    modifier onlyAdmin() {
        require(msg.sender == admin, "Access denied: only admin");
        _;
    }

    modifier onlyAuthorized() {
```

```
33          require(authorized[msg.sender], "Access denied:
               unauthorized user");
34          _;
35      }
36
37  constructor() {
38      admin = msg.sender; // Inicializa o administrador como o
           criador do contrato
39      authorized[admin] = true; // Admin j      autorizado por
           padr o
40  }
41
42
43      // Funcao para verificar se a data esta no formato DD/MM/YYYY
44      function isValidDate(string memory _dateOfBirth) internal view
           returns (bool) {
45          bytes memory dateBytes = bytes(_dateOfBirth);
46          // Deve ter exatamente 10 caracteres (DD/MM/YYYY)
47          if (dateBytes.length != 10) return false;
48
49          // Verifica a formatacao
50          for (uint i = 0; i < dateBytes.length; i++) {
51              if (i == 2 || i == 5) {
52                  if (dateBytes[i] != '/') return false; // Deve ser
                       uma barra
53              } else {
54                  if (dateBytes[i] < '0' || dateBytes[i] > '9')
                       return false; // Deve ser um digito
55              }
56          }
57
58          // Extrai o ano da data
59          uint year = (uint(parseInt(string(abi.encodePacked(
               dateBytes[6], dateBytes[7], dateBytes[8], dateBytes[9]))
               )));
60
61          // Obtem o ano atual
62          uint currentYear = block.timestamp / 31557600 + 1970; //
               Aproximacao do ano atual em segundos
63
64          // Verifica se o ano esta entre 1500 e o ano atual
65          if (year < 1500 || year >= currentYear) return false;
```

```
66
67          return true;
68      }
69
70      // Funcao auxiliar para converter string em uint
71      function parseInt(string memory _a) internal pure returns (uint
            ) {
72          bytes memory b = bytes(_a);
73          uint result = 0;
74          for (uint i = 0; i < b.length; i++) {
75              result = result * 10 + (uint8(b[i]) - 48); // Converte
                    caractere para inteiro
76          }
77          return result;
78      }
79
80      // Funcao para criar uma identidade de pessoa
81      function createPerson(
82          string memory _name,
83          string memory _dateOfBirth,
84          string memory _idDocument,
85          string memory _addressLocation,
86          string memory _phone,
87          string memory _nationality,
88          string memory _naturalness,
89          string memory _ethnicity
90      ) public {
91          require(!persons[msg.sender].exists, "Person already exists
                ");
92          require(isValidDate(_dateOfBirth), "Invalid date format,
                use DD/MM/YYYY");
93
94          persons[msg.sender] = Person({
95              name: _name,
96              dateOfBirth: _dateOfBirth,
97              idDocument: _idDocument,
98              addressLocation: _addressLocation,
99              phone: _phone,
100             nationality: _nationality,
101             naturalness: _naturalness,
102             ethnicity: _ethnicity,
103             exists: true
```

```
104         });
105
106         emit PersonCreated(msg.sender, _name);
107     }
108
109     // Funcao para atualizar dados de uma pessoa
110     function updatePerson(
111         string memory _name,
112         string memory _dateOfBirth,
113         string memory _idDocument,
114         string memory _addressLocation,
115         string memory _phone,
116         string memory _nationality,
117         string memory _naturalness,
118         string memory _ethnicity
119     ) public onlyAuthorized {
120         require(persons[msg.sender].exists, "Person does not exist"
                );
121         require(isValidDate(_dateOfBirth), "Invalid date format,
                use DD/MM/YYYY");
122
123         persons[msg.sender] = Person({
124             name: _name,
125             dateOfBirth: _dateOfBirth,
126             idDocument: _idDocument,
127             addressLocation: _addressLocation,
128             phone: _phone,
129             nationality: _nationality,
130             naturalness: _naturalness,
131             ethnicity: _ethnicity,
132             exists: true
133         });
134
135         emit PersonUpdated(msg.sender, _name);
136     }
137
138     // Funcao para verificar se a pessoa existe
139 function getPerson(address _user) public view onlyAuthorized
        returns (
140     string memory name,
141     string memory dateOfBirth,
142     string memory idDocument,
```

```
143     string memory addressLocation,
144     string memory phone,
145     string memory nationality,
146     string memory naturalness,
147     string memory ethnicity
148 ) {
149     require(authorized[msg.sender], "Access denied: unauthorized
            user");
150     require(persons[_user].exists, "Person not found");
151     Person storage person = persons[_user];
152     return (
153         person.name,
154         person.dateOfBirth,
155         person.idDocument,
156         person.addressLocation,
157         person.phone,
158         person.nationality,
159         person.naturalness,
160         person.ethnicity
161     );
162 }
163
164
165     // Funcao de autenticacao para autorizar usuarios
166     function grantAccess(address _user) public onlyAdmin {
167         require(!authorized[_user], "User already authorized");
168         authorized[_user] = true;
169         emit AccessGranted(_user);
170     }
171
172     // Funcao para revogar acesso de um usuario
173     function revokeAccess(address _user) public onlyAdmin {
174         require(authorized[_user], "User not authorized");
175         authorized[_user] = false;
176         emit AccessRevoked(_user);
177     }
178
179     // Verifica se um usuario tem acesso
180     function isAuthorized(address _user) public view returns (bool)
            {
181         return authorized[_user];
182     }
```

```
183 }
184         }
```

APÊNDICE B – Testes do contrato em Javascript

Listing B.1 – Testes do contrato em Javascript

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("IdentityManagement", function () {
  let IdentityManagement, identityContract, admin, user1, user2;

  beforeEach(async function () {
    // Deploy the contract and get signers
    IdentityManagement = await ethers.getContractFactory("
       IdentityManagement");
    [admin, user1, user2] = await ethers.getSigners();

    // Deploy the contract directly without calling .deployed()
    identityContract = await IdentityManagement.deploy();
  });

  it("O administrador deve ser capaz de criar sua pr pria
     identidade", async function () {
    await identityContract.connect(admin).createPerson(
      "Admin User",
      "01/01/1990",
      "123456",
      "Admin Address",
      "123-456-7890",
      "Nationality",
      "Naturalness",
      "Ethnicity"
    );

    const person = await identityContract.getPerson(admin.address);
    expect(person.name).to.equal("Admin User");
  });

```

```
32  it("O administrador deve ser capaz de atualizar sua pr pria
       identidade", async function () {
33    // Admin creates their own identity
34    await identityContract.connect(admin).createPerson(
35      "Admin User",
36      "01/01/1990",
37      "123456",
38      "Admin Address",
39      "123-456-7890",
40      "Nationality",
41      "Naturalness",
42      "Ethnicity"
43    );
44
45    // Now, admin updates their own identity
46    await identityContract.connect(admin).updatePerson(
47      "Updated Admin User",
48      "01/01/1985",
49      "654321",
50      "Updated Admin Address",
51      "321-654-0987",
52      "Updated Nationality",
53      "Updated Naturalness",
54      "Updated Ethnicity"
55    );
56
57    // Fetch the updated identity
58    const updatedPerson = await identityContract.getPerson(admin.
         address);
59
60    // Check that the values have been updated correctly
61    expect(updatedPerson.name).to.equal("Updated Admin User");
62    expect(updatedPerson.dateOfBirth).to.equal("01/01/1985");
63    expect(updatedPerson.idDocument).to.equal("654321");
64    expect(updatedPerson.addressLocation).to.equal("Updated Admin
         Address");
65    expect(updatedPerson.phone).to.equal("321-654-0987");
66    expect(updatedPerson.nationality).to.equal("Updated Nationality
         ");
67    expect(updatedPerson.naturalness).to.equal("Updated Naturalness
         ");
68    expect(updatedPerson.ethnicity).to.equal("Updated Ethnicity");
```

```
69  });
70
71
72      it("O usu rio1 n o deve conseguir acessar a identidade do
           administrador sem autoriza  o", async function () {
73      // User1 trying to access Admin's identity should fail with "
           Access denied: unauthorized user"
74      await expect(identityContract.connect(user1).getPerson(admin.
           address))
75          .to.be.revertedWith("Access denied: unauthorized user");
76      });
77
78      it("O administrador deve ser capaz de revogar o acesso do
           usu rio2", async function () {
79          // Admin cria sua pr pria identidade
80          await identityContract.connect(admin).createPerson(
81            "Admin User",
82            "01/01/1990",
83            "123456",
84            "Admin Address",
85            "123-456-7890",
86            "Nationality",
87            "Naturalness",
88            "Ethnicity"
89          );
90
91          // Admin concede acesso ao User2
92          await identityContract.connect(admin).grantAccess(user2.
               address);
93
94          // Verifique se User2 pode acessar a identidade do Admin
95          await expect(identityContract.connect(user2).getPerson(
               admin.address))
96            .to.not.be.reverted;
97
98          // Agora, Admin revoga o acesso do User2
99          await identityContract.connect(admin).revokeAccess(user2.
               address);
100
101         // User2 n o deve mais conseguir acessar a identidade do
               Admin
```

```
102         await expect(identityContract.connect(user2).getPerson(
                admin.address))
103           .to.be.revertedWith("Access denied: unauthorized user");
104       });
105
106
107
108
109   it("O administrador deve conceder acesso ao Usu rio2 e o
        Usu rio2 deve poder acessar a identidade do administrador",
        async function () {
110     // Admin creates their own identity
111     await identityContract.connect(admin).createPerson(
112       "Admin User",
113       "01/01/1990",
114       "123456",
115       "Admin Address",
116       "123-456-7890",
117       "Nationality",
118       "Naturalness",
119       "Ethnicity"
120     );
121
122     // Admin grants access to User2
123     await identityContract.connect(admin).grantAccess(user2.address
            );
124
125     // User2 should now be able to access Admin's identity
126     await expect(identityContract.connect(user2).getPerson(admin.
            address))
127       .to.not.be.reverted;
128   });
129
130
131   it("O usu rio1  n  o deve ser capaz de atualizar a identidade sem
        autoriza  o", async function () {
132     // Try updating as User1 without authorization
133     await expect(identityContract.connect(user1).updatePerson(
134       "User1 Name",
135       "02/02/1995",
136       "654321",
137       "User1 Address",
```

```
138         "987-654-3210",
139         "Other Nationality",
140         "Other Naturalness",
141         "Other Ethnicity"
142       )).to.be.revertedWith("Access denied: unauthorized user");
143     });
144 });
```