DOCUMENTAÇÃO TÉCNICA

Experimento de laboratório: Uso de blockchain para o armazenamento de dados na microchipagem animal

Autoria: GISELE BRANDAO KANDA, MARCELO AOKI

SUMÁRIO

1. INTRODUÇÃO

Este experimento de laboratório empregou a linguagem de programação Python, na versão 3.10, para desenvolver a solução.

O projeto de *blockchain* compreendeu cinco principais componentes interdependentes, cada um desempenhando um papel na estrutura da rede. Esses componentes são o "Microchip Animal," a "Entidade Veterinária," as "Transações", os "Blocos" e a própria "Blockchain".

- Microchip animal: este componente incorpora um código de identificação único, servindo como a identidade digital de um animal de estimação, a quem os dados inseridos na blockchain se referem.
- Entidade veterinária: é o profissional da área ou empresa que assume a responsabilidade pela inserção, edição e consulta dos dados associados a um animal no sistema. Ela desempenha um papel fundamental na integridade e precisão das informações.
- Transações: neste contexto, as "Transações" representam os dados que serão registrados na blockchain. O termo é adotado em conformidade com a tradição blockchain de se referir aos dados como "transações," embora esses dados abranjam informações variadas, como detalhes de propriedade, históricos médicos e eventos relevantes.
- Blocos: cada um dos "Blocos" na blockchain é um recipiente que agrupa um conjunto de transações.
- Blockchain: a "Blockchain", propriamente dita, é a infraestrutura que abriga todos os blocos de transações. Os blocos são ligados entre si em uma cadeia sequencial (figurativamente conhecida como “corrente”), criando assim um registro cronológico de todas as transações realizadas.

2. IMPORTAÇÃO DE PACOTES

```
import json
import hashlib
from datetime import datetime
from uuid import uuid1
```

- **`json`**: usado para lidar com dados no formato JSON (*JavaScript Object Notation*).
- **`hashlib`**: fornece funcionalidades para cálculos de *hashes* criptográficos.
- **`datetime`**: usado para lidar com datas e horas.

- **`uuid1`**: O módulo `uuid1` faz parte do pacote `uuid`, que é usado para gerar identificadores únicos universais (UUIDs). Um UUID é uma sequência de 128 bits que é garantida como única em um espaço de tempo e espaço específicos. O `uuid1` gera UUIDs baseados no horário e no endereço MAC da máquina, tornando-os adequados para identificadores únicos.

3. CLASSE MICROCHIP

```
class Microchip:

    def __init__(self, nome, genero, raca, nascimento):
        self.id = str(uuid1())
        self.nome = nome
        self.genero = genero
        self.raca = raca
        if self.validar_data(nascimento):
            self.nascimento = nascimento
        else:
            raise ValueError("Data de nascimento inválida.")

    def validar_data(self, date_str):
        try:
            datetime.strptime(date_str, "%Y-%m-%d")
            return True
        except ValueError:
            return False

    def consultar_dados(self):
        return self.__dict__
```

- A classe `Michochip` é definida.
- No método `__init__` (o construtor da classe), a classe aceita quatro parâmetros: `nome`, `genero`, `raca` e `nascimento`. O construtor é usado para inicializar os atributos do objeto. Aqui estão as ações realizadas no construtor:
    - O `self.id` é gerado com base no UUID (identificador único universal) usando a função uuid1(). Isso cria um identificador único para cada objeto `Michochip`.
    - O `self.nome`, `self.genero`, `self.raca` e `self.nascimento` são inicializados com os valores passados como argumentos para o construtor, representando o nome, gênero, raça e data de nascimento do animal.

- O construtor também verifica se a data de nascimento fornecida é válida, chamando o método `self.validar_data(nascimento)`. Se a data não for válida, ele lança uma exceção `ValueError` com a mensagem "Data de nascimento inválida".

- O método `validar_data` verifica se a string data é válida. Ele analisa a string data usando o formato "%Y-%m-%d" (ano-mês-dia) com a função `datetime.strptime`. Se a conversão for bem-sucedida, a data é considerada válida e o método retorna `True`. Caso contrário, se a conversão causar uma exceção ValueError, o método retorna `False`.
- O método `consultar_dados` é usado para retornar um dicionário com todos os dados do objeto `Michochip`.

4. CLASSE ENTIDADE VETERINÁRIA

```
class EntidadeVeterinaria:

    def __init__(self, nome, tipo, localizacao):
        self.id = str(uuid1())
        self.nome = nome
        self.tipo = tipo
        self.localizacao = localizacao

    def consultar_dados(self):
        return self.__dict__
```

- A classe `EntidadeVeterinaria` é definida.
- No método `__init__` (o construtor da classe), a classe aceita três parâmetros: `nome`, `tipo` e `localização`. O construtor é usado para inicializar os atributos do objeto. Aqui estão as ações realizadas no construtor:
    - O `self.id` é gerado com base em um UUID (Identificador Único Universal) usando a função `uuid1()`. Isso cria um identificador único para cada objeto `EntidadeVeterinaria`.
    - O `self.nome, self.tipo` e `self.localizacao` são inicializados com os valores passados como argumentos para o construtor, representando o nome, tipo e localização física da entidade. O tipo pode ser "Veterinário", "Clínica Veterinária", “Canil” etc.

- O método `consultar_dados` é usado para retornar um dicionário com todos os dados do objeto `EntidadeVeterinaria`.

5. CLASSE TRANSAÇÃO

Informações de uma transação:

Obrigatoriedade	**Campo**	**Descrição**
obrigatório	entidade	ID da entidade veterinária que está criando a transação.
	microchip	ID do microchip que representa o animal cuja informação se deseja registrar.
	dados	Dicionário com os dados que se deseja registrar acerca do animal.

Dicionário que o campo ‘dados’ aceita:

Obrigatoriedade	**Campo**	**Descrição**
obrigatório	data	Data da ocorrência da informação que se quer registrar.
opcional	propriedade	Dicionário com informações da nova propriedade, nos casos de transferência.
	registro_medico	Informação textual sobre um novo dado de registro médico do animal (ex.: cirurgias).
	observacoes	Informação textual sobre uma nova informação do animal (ex.: pedigree).

```
class Transacao:

    def __init__(self, entidade, microchip, dados):
        self.id = str(uuid1())
        self.entidade = entidade
        self.microchip = microchip

        campos_permitidos = {'data', 'propriedade',
                                'registro_medico', 'observacoes'}
        for campo in dados:
            if campo not in campos_permitidos:
                raise ValueError(f"O campo '{campo}' não é permitido
```

```
                                                        nos dados.")

        if 'data' in dados and self.validar_data(dados['data']):
            self.dados = {'data': dados['data']}
        else:
            raise ValueError("O campo 'data' é obrigatório e deve
                                    estar no formato YYYY-MM-DD.")

        if 'propriedade' in dados:
            self.dados['propriedade'] =
                      self.validar_propriedade(dados['propriedade'])
        else:
            self.dados['propriedade'] = ''

        self.dados['registro_medico'] =
                                    dados.get('registro_medico', '')
        self.dados['observacoes'] = dados.get('observacoes', '')


    def validar_data(self, date_str):
        try:
            datetime.strptime(date_str, "%Y-%m-%d")
            return True
        except ValueError:
            return False

    def validar_propriedade(self, propriedade):
        campos_permitidos = {'proprietario', 'endereco', 'email'}
        for campo in propriedade:
            if campo not in campos_permitidos:
                raise ValueError(f"O campo 'propriedade/{campo}' não
                                            é permitido nos dados.")

        if 'proprietario' in propriedade and
           'endereco' in propriedade and
           'email' in propriedade:
            return {
                'proprietario': propriedade['proprietario'],
                'endereco': propriedade['endereco'],
                'email': propriedade['email']
            }
        else:
            raise ValueError("Os campos 'proprietario', 'endereco' e
                         'email' são obrigatórios em 'propriedade.")

    def consultar_dados(self):
        dados = {
            'id': self.id,
```

```
            'entidade': self.entidade,
            'microchip': self.microchip,
            'dados': {
                'data': self.dados['data'],
                'propriedade': self.dados['propriedade'],
                'registro_medico': self.dados['registro_medico'],
                'observacoes': self.dados['observacoes']
            }
        }
        return dados
```

- A classe `Transacao` é definida.
- No método `__init__` (o construtor da classe), a classe aceita três parâmetros: `entidade`, `microchip` e `dados`. O construtor é usado para inicializar os atributos do objeto. Aqui estão as ações realizadas no construtor:
    - Verifica se os campos em `dados` são permitidos, ou seja, se eles correspondem a "data", "propriedade", "registro_medico" e "observacoes". Se houver algum campo não permitido, uma exceção é levantada.
    - Verifica a presença e o formato da data. Se a data for válida no formato "YYYY-MM-DD", ela é armazenada no dicionário `self.dados`.
    - Verifica a propriedade, garantindo que os campos "proprietario", "endereco" e "email" estejam presentes. Se estiverem, essas informações são armazenadas no dicionário `self.dados`.
    - Define os campos `registro_medico` e `observacoes` com base nos valores fornecidos em `dados`.
- O método `validar_data` é usado para verificar se a data fornecida está no formato correto "YYYY-MM-DD". Ele retorna `True` se a data for válida e `False` caso contrário.
- O método `validar_propriedade` valida as informações da propriedade, garantindo que os campos obrigatórios "proprietario", "endereco" e "email" estejam presentes. Se estiverem, essas informações são armazenadas em um dicionário.
- O método `consultar_dados` retorna um dicionário com informações da transação, incluindo “id”, “entidade”, “microchip” e os “dados” validados armazenados no dicionário `self.dados`.

5.1. TRANSAÇÕES PENDENTES

É importante esclarecer que as transações não são inseridas imediatamente em um bloco da blockchain. Ao invés disso, elas são submetidas a um processo de mineração, no qual são verificadas, agrupadas e registradas em blocos subsequentes.

As transações recém-criadas são inseridas em uma fila de "Transações Pendentes", aguardando que um novo bloco seja minerado. À medida que novos blocos são adicionados à blockchain, as transações são confirmadas e se tornam parte do registro público. Para criar uma fila, foi declarado uma variável do tipo lista chamada `transacoes_pendentes` da seguinte forma:

```
transacoes_pendentes = [ ]
```

A seguir, apresenta-se um código que ilustra o uso da lista de transações pendentes:

```
t1 = Transacao(entidade1.id, microchip1.id, dados1)
transacoes.append(t1)

t2 = Transacao(entidade2.id, microchip2.id, dados2)
transacoes.append(t2)
```

6. CLASSE BLOCO

```
class Bloco:

    def __init__(self, transacoes, hash_anterior):
        self.transacoes = transacoes
        self.timestamp = datetime.now().isoformat()
        self.hash = self.calcular_hash()
        self.hash_anterior = hash_anterior

    def calcular_hash(self):
        dados = json.dumps(self.__dict__, sort_keys=True)
        return hashlib.sha256(dados.encode()).hexdigest()
```

- A classe `Bloco` é definida.

- O construtor `def __init__(self, transacoes, hash_anterior)` da classe `Bloco`. Ele recebe dois parâmetros `transacoes (transacoes)`, que representa uma lista de objetos `Transacao` e o `hash_anterio`r (hash do bloco anterior).
- O `self.transacoes = transacoes` atribui valor ao atributo `transacoes` do objeto `Bloco`.
- A classe `self.timestamp = datetime.now().isoformat()` é usada para obter a data e hora atual e, em seguida, `isoformat()` é chamado para obter uma representação em formato de string no padrão ISO 8601, que é vinculada ao atributo timestamp do objeto `Bloco`, como objetivo de registrar quando o bloco foi construído.
- O trecho `self.hash = self.calcular_hash()` chama o método `calcular_hash()` para calcular o hash de bloco e atribuí-lo ao hash do objeto.
- O `self.hash_anterior = hash_anterior` o valor é atribuído ao `hash_anterior` do objeto `Bloco`, que representa o hash do bloco anterior na blockchain.
- O `def calcular_hash(self)` é um método que é usado para calcular o hash do bloco. Ele retorna o hash SHA-256 dos dados do bloco.
- O `dados = json.dumps(self.__dict__, sort_keys=True)` e o método `json.dumps()` são usados para converter o dicionário contendo os atributos do objeto `Bloco` em uma string JSON. O argumento `sort_keys=True` garante que as chaves do dicionário sejam ordenadas alfabeticamente antes de gerar uma string JSON, tendo em vista que a ordem das chaves implica em um hash diferente no final.
- O `return hashlib.sha256(dados.encode()).hexdigest()` cria um objeto `hashlib` que calcula o hash SHA-256 da string JSON representando os dados do bloco. O método `encode()` é usado para converter a string em bytes antes de calcular o hash. Em seguida, o método `hexdigest()` é chamado para obter a representação hexadecimal do hash, que é retornada como o resultado.

6.1. MINERAÇÃO DO BLOCO E PROVA DE TRABALHO

```
class Bloco:

    def __init__(self, transactions, hash_anterior):
        ...
        self.nonce = 0
```

```
  def calcular_hash(self):
       ...

   def prova_de_trabalho(self, dificuldade):
       while self.hash[:dificuldade] != '0' * dificuldade:
           self.nonce += 1
           self.hash = self.calcular_hash()

   def minerar_bloco(self, dificuldade):
       self.prova_de_trabalho(dificuldade)
       print("Bloco minerado com sucesso! Nonce:", self.nonce)
```

- O `self.nonce = 0` adicionou um atributo nonce ao bloco, que é iniciado com o valor 0. O nonce é um número arbitrário usado na prova de trabalho para encontrar um hash válido.
- O `def prova_de_trabalho(self, dificuldade)` é um novo método que implementa a prova de trabalho. A prova de trabalho é feita através de um loop enquanto o hash do bloco não satisfaz a dificuldade especificada (ou seja, não começa com um certo número de zeros no início). O loop incrementa o valor do nonce e recalcula o hash até encontrar um hash válido.
- O método `def minerar_bloco(self, dificuldade)` é usado para minerar um bloco. Ele chama o método `prova_de_trabalho()` com a dificuldade especificada e, quando um hash válido é encontrado, imprime uma mensagem indicando que o bloco foi minerado com sucesso, junto com o valor do nonce que levou à obtenção desse hash válido.

7. CLASSE BLOCKCHAIN

```
class Blockchain:
    def __init__(self):
        self.chain = []
        self.transacoes_pendentes = []
        self.dificuldade = 4
```

- O `self.chain` é a lista que representa a cadeia de blocos na blockchain. Inicialmente, essa lista está vazia, mas em seguida será construído bloco gênesis. Cada elemento dessa lista será um objeto da classe `Bloco`, contendo informações sobre as transações e o hash do bloco.

- O `self.transacoes_pendentes` é uma lista e armazena as transações que ainda não foram incluídas no bloco da blockchain. Quando as transações são enviadas para a blockchain, elas são inicialmente colocadas nesta lista e, posteriormente, serão incluídas em um bloco quando houver transações suficientes para criar um bloco.
- O `self.dificuldade` é um valor numérico que representa a dificuldade da prova de trabalho para minerar um novo bloco. Neste caso, a dificuldade é definida como 4, o que significa que o hash do bloco minerado deve começar com quatro zeros (0000) para ser considerado válido. Quanto maior a dificuldade, maior é o tempo de mineração do bloco, o que ajuda a garantir segurança a blockchain.

7.1. BLOCO GÊNESIS

```
class Blockchain:
    def __init__(self):
        self.chain = [self.bloco_genesis()]
        ...

    def bloco_genesis(self):
        transacao_genesis = Transacao("0", "0", {
            "data": datetime.now().strftime("%Y-%m-%d"),
            "observacoes": "Este é o bloco Gênesis"})
        return Bloco([transacao_genesis], hash_anterior="0")
```

- O `self.chain = [self.bloco_genesis()` é parte do construtor `__init__` da classe `Blockchain`. Quando uma instância da classe `Blockchain` é criada, a primeira coisa que acontece é a inicialização da variável `self.chain`. Neste caso, `self.chain` é definida como uma lista que contém o resultado da função `self.bloco_genesis()`. Isso significa que a cadeia de blocos começa com um único bloco, o bloco de gênesis.
- O `def bloco_genesis(self)` é um método dentro da classe `Blockchain` que é usada para construir o bloco de gênesis.
- O `transacao_genesis = Transacao("0", "0", { ... })` constrói uma instância da classe `Transacao` para representar a transação dentro do bloco de gênesis. Os argumentos passados são "0" para a entidade, "0" para o microchip e um dicionário de dados que contém a data atual e uma observação.

- O `return Bloco([transacao_genesis], hash_anterior="0")` um novo bloco é construído usando a classe `Bloco`. O bloco contém uma lista de transações, que no caso do bloco de gênesis, contém apenas a transação de gênesis. O hash_anterior é definido como "0", indicando que este é o primeiro bloco na cadeia e não há blocos anteriores.

7.2. ADIÇÃO DE BLOCOS

```
    def adicionar_bloco(self):
        if len(self.transacoes_pendentes) < 3:
            raise ValueError("Não há transações suficientes para
                                        criar um novo bloco.")

        transacoes_para_incluir = self.transacoes_pendentes[:3]
        novo_bloco = Bloco(transacoes_para_incluir,
                                         self.chain[-1].hash)
        novo_bloco.minerar_bloco(self.dificuldade)

        self.chain.append(novo_bloco)
        self.transacoes_pendentes = self.transacoes_pendentes[3:]
```

- O `def adicionar_bloco(self)` definição do método para adicionar um novo bloco à blockchain.
- O `if len(self.transacoes_pendentes) < 3` verifica se existe pelo menos 3 transações pendentes para criar um novo bloco. Se houver menos de 3 transações, uma exceção `ValueError` é levantada, indicando que não há transações suficientes para criar um novo bloco.
- O `transacoes_para_incluir = self.transacoes_pendentes[:3]` indica que se houver ao menos 3 transações pendentes, o código seleciona as três primeiras transações da lista `self.transacoes_pendentes` e as armazena em uma nova lista chamada `transacoes_para_incluir`. Essas são as transações que serão incluídas no novo bloco.
- O `novo_bloco = Bloco(transacoes_para_incluir, self.chain[-1].hash)` nesta linha um novo bloco é construído usando a classe `Bloco`. Ele recebe as transações selecionadas em `transacoes_para_incluir` e o hash do bloco anterior (o último bloco na cadeia) como seu `hash_anterior`.

- O `novo_bloco.minerar_bloco(self.dificuldade)` minera um novo bloco com a dificuldade especificada pela variável `self.dificuldade`. O processo da prova de trabalho é ajustado iterativamente até que seu hash atenda aos critérios de dificuldade definidos.
- O `self.chain.append(novo_bloco)` adicona um novo bloco o à cadeia de blocos existente, tornando-se o último bloco na blockchain.
- O `self.transacoes_pendentes = self.transacoes_pendentes[3:]` inclui as três primeiras transações no novo bloco e são removidas da lista `self.transacoes_pendentes`, já que agora foram processadas e incluídas em um bloco. Isso garante que apenas as transações não processadas permaneçam na lista de transações pendentes.

7.3. VERIFICAÇÃO DA BLOCKCHAIN

```
def verificar_blockchain(self):
     for i in range(1, len(self.chain)):
         bloco_atual = self.chain[i]
         bloco_anterior = self.chain[i - 1]

         if bloco_atual.hash !=
             bloco_atual.calcular_hash():
                   return False

         if bloco_atual.hash_anterior !=
             bloco_anterior.hash:
                   return False

     return True
```

- O `def verificar_blockchain(self)` define o método de verificação.
- O `for i in range(1, len(self.chain))` o loop itera através dos blocos na cadeia, começando do segundo bloco "(índice 1)" até o último bloco na lista `self.chain`.
- O `bloco_atual = self.chain[i] e bloco_anterior = self.chain[i - 1]` são definidas duas variáveis, dentro do loop, bloco_atual e bloco_anterior, que

representam o bloco atual sendo verificado e o bloco anterior na cadeia, respectivamente.

- O `if bloco_atual.hash != bloco_atual.calcular_hash()` verifica se o hash do bloco atual `(bloco_atual.hash)` corresponde ao hash real calculado do bloco `(bloco_atual.calcular_hash())`. Se houver uma discrepância entre o hash armazenado no bloco e o hash real, a função retorna `False`, indicando que a blockchain está comprometida.
- O `if bloco_atual.hash_anterior != bloco_anterior.hash` verifica se o hash anterior do bloco atual `(bloco_atual.hash_anterior)` corresponde ao hash do bloco anterior `(bloco_anterior.hash)`. Se houver uma diferença, a função retorna `False`.
- O `return True` verifica se nenhum problema foi encontrado nos blocos, o método retorna `True`. Indicando que a blockchain não foi violada e todos os blocos estão corretamente encadeados.

8. EXEMPLOS DE USO

a) Criação da blockchain

```
petcoin = Blockchain()
```

b) Adição na lista de transações pendentes

```
# Criar uma transação
transacao1 = Transacao("Entidade1", "Microchip1", {
    "data": "2023-10-02"),
    "observacoes": "Primeira transação pendente"
})

# Adicionar a transação à lista de transações pendentes
blockchain.transacoes_pendentes.append(transacao1)
```

c) Consulta de transações pendentes

```
print("Transações Pendentes:")
for transacao in blockchain.transacoes_pendentes:
    print(f"ID da Transação: {transacao.id}")
```

d) Adição do novo bloco

```
# Verificar se há transações pendentes suficientes
if len(blockchain.transacoes_pendentes) >= 3:
    # Adicionar um novo bloco à blockchain
    blockchain.adicionar_bloco()
    print("Novo bloco adicionado à blockchain!")
```

e) Consulta do bloco gênesis

```
bloco_geneses = blockchain.chain[0]

# Exibir informações do bloco de gênesis
print("Bloco de Gênesis:")
print(f"Timestamp: {bloco_geneses.timestamp}")
print("Transações:")
for transacao in bloco_geneses.transacoes:
    print(f"  ID da Transação: {transacao.id}")
```

f) Consulta do último bloco

```
ultimo_bloco = blockchain.chain[-1]

# Exibir informações do último bloco
print("Último Bloco na Blockchain:")
print(f"Timestamp: {ultimo_bloco.timestamp}")
print("Transações:")
for transacao in ultimo_bloco.transacoes:
    print(f"  ID da Transação: {transacao.id}")
```

g) Iteração por todos os blocos

```
for i, bloco in enumerate(blockchain.chain):
    print(f"Bloco {i + 1}:")
    print(f"Timestamp: {bloco.timestamp}")
    print(f"Hash do Bloco: {bloco.hash}")
    print(f"Hash Anterior: {bloco.hash_anterior}")
    print("Transações:")
    for transacao in bloco.transacoes:
        print(f"  ID da Transação: {transacao.id}")
    print()
```

h) Função que busca transações de um determinado microchip

```
def buscar_transacoes_por_microchip(blockchain, microchip):
    transacoes_relacionadas = []
    for bloco in blockchain.chain:
        for transacao in bloco.transacoes:
            if transacao.microchip == microchip:

     transacoes_relacionadas.append(
                                transacao.consultar_dados())
     return transacoes_relacionadas
```

i) Função que busca transações por uma determinada entidade veterinária

```
def buscar_transacoes_por_entidade(blockchain, entidade):
    transacoes_relacionadas = []
    for bloco in blockchain.chain:
        for transacao in bloco.transacoes:
            if transacao.entidade == entidade:

     transacoes_relacionadas.append(
                                transacao.consultar_dados())
     return transacoes_relacionadas
```