UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

JURANDI DALEFFE JUNIOR

ANÁLISE DE DESENVOLVIMENTO DE UMA APLICAÇÃO BASEADA EM UM FRAMEWORK BLOCKCHAIN

PATO BRANCO

2022

JURANDI DALEFFE JUNIOR

ANÁLISE DE DESENVOLVIMENTO DE UMA APLICAÇÃO BASEADA EM UM FRAMEWORK BLOCKCHAIN

Development analysis of an application based on a blockchain framework

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Prof. Me. Adriano Serckumecka

PATO BRANCO

2022

JURANDI DALEFFE JUNIOR

ANÁLISE DE DESENVOLVIMENTO DE UMA APLICAÇÃO BASEADA EM UM FRAMEWORK BLOCKCHAIN

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de Aprovação: 23 de junho de 2022.

Prof. Me. Adriano Serckumecka
Universidade Tecnológica Federal do Paraná

Prof. Dr. Érick Oliveira Rodrigues
Universidade Tecnológica Federal do Paraná

Prof. Dr. Fábio Favarim
Universidade Tecnológica Federal do Paraná

PATO BRANCO

2022

AGRADECIMENTOS

Agradeço a todos que colaboraram para o desenvolvimento deste trabalho, em especial aos meus pais, Margarete Girardi Daleffe e Jurandi Daleffe, que me deram a oportunidade de estar aqui hoje. Agradeço também a minha namorada, Bianca Andressa Cardoso, que me auxiliou e me deu apoio durante todo o percurso até aqui. Agradeço também ao Professor Adriano Serckumecka por todo seu auxílio durante o desenvolvimento deste trabalho como professor orientador. E agradeço também a banca que disponibilizou seu tempo para conclusão desta etapa que é um marco na minha vida.

Os eruditos são aqueles que leram nos livros; mas os pensadores, os gênios, os iluminadores do mundo e os promotores do gênero humano são aqueles que leram diretamente no livro do mundo.

Arthur Schopenhauer

RESUMO

O objetivo deste trabalho é realizar um comparativo entre *frameworks* da rede *Blockchain* trazendo uma tabela comparativa contemplando informações sobre seu método de funcionamento, capacidade e escalabilidade. Além de auxiliar novos usuários, que tenham interesse no quesito de programação para *Blockchain*, em como executar uma aplicação *Hyperledger Fabric* utilizando também de um guia para desenvolvimento modelado a partir da biblioteca *CC-Tools*.

Palavras-chave: Blockchain. Framework. Hyperledger. Comparativo. Desempenho.

ABSTRACT

The objective of this project is to make a comparison among frameworks of the Blockchain network bringing a comparative table contemplating information about it's method of operation, capacity and scalability. In addition to helping new users, who are interested in Blockchain programming, and also on how to run a Hyperledger Fabric application, using a development guide modeled for beginner from the CC-Tools library.

Keywords: Blockchain. Framework. Hyperledger. Comparation. Performance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGEM DE CÓDIGOS FONTE

LISTA DE ABREVIATURAS E SIGLAS

SIGLAS

CA	Certificado de autenticação, do inglês *certification authorities*
CPF	Cadastro de pessoa física
EVM	Máquina virtual do Ethereum, do inglês *Ethereum virtual machine*
FBA	Acordo Bizantino Federado, do inglês *Federated Byzantine Agreement*
IoT	Internet das coisas, do inglês *Internet of things*
MSP	Provedor de serviço de associado, do inglês *Membership Service Provider*
P2P	Par a par, do inglês *peer-to-peer*
pBFT	Tolerância de Falha Bizantina Prática, do inglês *Practical Byzantine Fault Tolerance*
PoS	Prova por trabalho, do inglês *Proof of work*
PoW	Prova por estocagem, do inglês *Proof of stake*
REST	Transferência do estado representacional, ou do inglês *Representational State Transfer*
TCP	Protocolo de controle de transmissão, do inglês *Transmission control protocol*
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO

Com o grande crescimento de popularidade da rede *Blockchain*, principalmente no âmbito das criptomoedas, é notável que algumas outras áreas derivativas dela comecem a ser notadas e estudadas para utilização em indústrias e grandes empresas. Diversos *frameworks* têm surgido e vêm sendo utilizados para as mais diversas aplicações. No entanto, podendo gerar dúvidas para quem está iniciando os estudos ou o desenvolvimento de aplicações baseadas nessa tecnologia.

Um dos *frameworks* que vem ganhando bastante destaque é conhecido por *Hyperledger*, mantido pela IBM, o qual possui várias subestruturas com ferramentas e bibliotecas que permitem implantar a rede *Blockchain* tanto em ambientes controlados como em produção. A IBM disponibiliza em seu site vários exemplos de aplicações em produção ou em estudo, baseadas nesse tipo de infraestrutura. É possível citar como exemplo o *The Home Depot*[1], um projeto para tornar o comércio entre fronteiras mais fácil e outro focado na conexão entre produtores e consumidores.

Portanto, este trabalho tem como foco principal a utilização de um framework Blockchain para o desenvolvimento de aplicações, diferente do que comumente é associado ao nome Blockchain, que são as criptomoedas. Após uma análise e comparativo dos principais frameworks disponíveis e mais utilizados no mercado, uma aplicação de uso geral foi proposta, bem como elaborado um manual de apoio para auxiliar os iniciantes nessa área, que frequentemente desconhecem os passos iniciais de como implementar algo utilizando essas tecnologias.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem como objetivo geral o desenvolvimento de uma aplicação utilizando um *framework Blockchain*, bem como gerar um material de apoio para iniciantes nesse tipo de tecnologia, detalhando os passos para o seu desenvolvimento.

1.1.2 Objetivos Específicos

Para alcançar o objetivo geral, os seguintes objetivos específicos foram definidos:

- Comparativo entre os principais *frameworks Blockchain*;
- Análise e desenvolvimento da aplicação *Blockchain*;
- Guia de desenvolvimento para aplicação proposta.

[1] https://www.ibm.com/br-pt/topics/hyperledger

1.2 JUSTIFICATIVA

Por ser uma tecnologia recente, apesar de já existir boa documentação sobre o tema, o campo de desenvolvimento de aplicações baseadas em *Blockchain* ainda é escasso, principalmente aos iniciantes, que sequer sabem qual o *framework* será melhor aplicado no seu caso.

Por isso, foi proposto uma aplicação genérica e introdutória baseada em *Blockchain*, demonstrando sua estrutura e detalhes necessários à sua implementação. Enfim, foi pretendido trazer mais visibilidade para a rede *Blockchain* em áreas que não englobam somente o contexto das criptomoedas, além de auxiliar e enriquecer o conhecimento de quem pretenda criar uma aplicação relacionada.

2 REFERENCIAL TEÓRICO

A área de criptomoedas vem ganhado bastante destaque nos últimos anos, porém a área de análise deste trabalho não é a rede de criptomoedas, e sim outra rede derivada da *Blockchain* que são seus *frameworks* para desenvolvimento de aplicações.

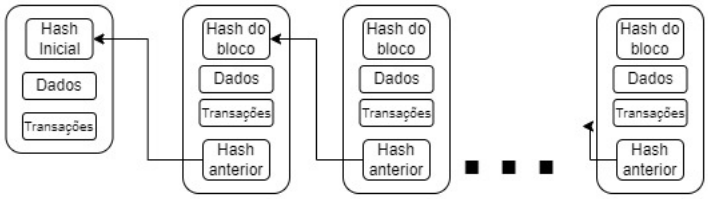
Resumidamente, um *framework* da rede *Blockchain*, é um conjunto de ferramentas e bibliotecas utilizadas para realizar a implantação dessa rede. Um dos *frameworks* mais conhecidos é o *Hyperledger*, sendo descrito como "[...]uma comunidade de código aberto focada no desenvolvimento de um conjunto de estruturas estáveis, ferramentas e bibliotecas para implantações de Blockchain de nível empresarial." Hyperledger (2020).

Com base nestes *frameworks*, uma aplicação utilizando a rede *Blockchain* pode ser desenvolvida tanto em nível experimental quanto comercial. Para que a aplicação produzida tenha um bom desempenho é necessário entender as características dos *frameworks* e verificar qual deles obterá o melhor resultado para o projeto em específico.

2.1 *BLOCKCHAIN*

Blockchain é uma série de bloco de dados relacionados, em que para um novo bloco ser inserido ele deve possuir relação com o anterior. Esta série de blocos é resistente a modificações pois ao alterar um bloco na rede, todos os blocos seguintes devem ser alterados também pois possuem uma relação de dependência entre si. Seu conceito foi introduzido em conjunto da criação da criptomoeda *Bitcoin*, que utiliza desta rede encadeada de dados como base de suas operações financeiras para transferência de posse do seu criptoativo.

Figura 1 – Diagrama de funcionamento dos blocos em uma *Blockchain*



Fonte: Autoria própria

É mostrado na Figura 1 que cada bloco em uma rede *Blockchain* é composto pelas transações confirmadas, por um *hash* do bloco e um *hash* do bloco anterior. Os valores de *hashes* são derivados de uma função unidirecional (função em que é executado um cálculo é fácil e o caminho inverso do mesmo possui uma complexidade muito alta) na qual o valor de entrada possui tamanho arbitrário e o valor de saída possui tamanho fixo Stevens *et al.* (2007).

Frequentemente associada a criptomoedas, a rede *Blockchain* é definida por Yaga *et al.* (2019) como livros contábeis ou livros razão[1](também chamados de *ledger*) resistentes a altera-

[1] um livro razão na economia é uma forma de registro cronológico de transações.

ções que não possuem um repositório central e não são controlados por uma autoridade central como bancos ou governos. Este livro razão, como pode ser chamado, permite que usuários registrem dados de forma permanente e inalterável após serem publicados.

Um complemento da definição de *Blockchain* dado por Sajana, Sindhu e Sethumadhavan (2018) é que a atualização da *Blockchain* é dada por um protocolo de consenso, o que garante uma ordenação inequívoca das suas transações. Com isso os blocos da rede garantem sua integridade e consistência, já que ao realizar uma alteração em um bloco todos os subsequentes seriam alterados. Outro fator relevante é que diferentes aplicações na rede *Blockchain* podem utilizar diferentes protocolos de consenso (as vezes mais de um), como é melhor explicado na subseção 2.1.2.

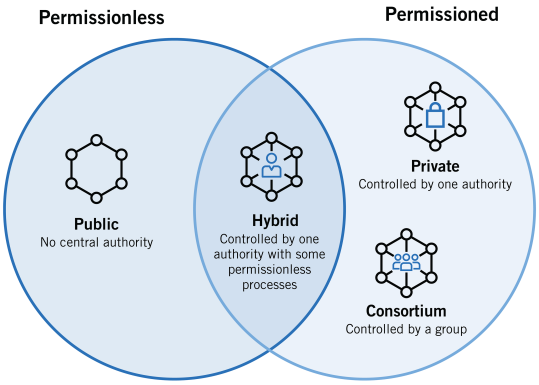
Desde sua criação a *Blockchain* vem evoluindo, estando agora em sua terceira geração, também chamada de *Blockchain* 3.0. Esta geração possui recursos que chamam a atenção, como aplicações com iteração entre máquinas, compatibilidade com dispositivos móveis e também protocolos de computação Tavares, Correia e Restivo (2019). Sua tendência é continuar evoluindo e englobando mais características de forma que suas aplicações possam ser utilizadas em diversas áreas.

2.1.1 Tipos de *Blockchain*

A rede *Blockchain* pode ser subdividida em 3 tipos, sendo elas: redes públicas, redes privadas e redes de consórcio ou federadas (DATTANI; SHETH, 2019). As redes públicas são aquelas em que qualquer pessoa pode ter acesso, podendo executar um nó e até mesmo um software para mineração. As *Blockchains* públicas possuem o código aberto (*open source*) nas quais qualquer pessoa pode verificar suas funcionalidades. Geralmente este modelo de *Blockchain* está associado a criptomoedas pois, qualquer pessoa tem a possibilidade de criar uma carteira e realizar a transferência de ativos. As redes privadas partem de um sistema fechado em que as pessoas não possuem a possibilidade de interagir caso não possuam as permissões necessárias. Geralmente é um tipo de *Blockchain* criado por pessoas em âmbito particular ou organizações, nos quais nesses casos há uma autoridade com o encargo de administrar as permissões. Os mecanismos de consenso deste tipo de *Blockchain* podem ser os mesmos utilizados em uma rede pública, sua maior diferença é a característica de acesso. As *Blockchains* de consórcio ou federadas são redes nas quais o poder de controle é dado a grupos de pessoas que formam consórcios. Ou seja, o poder é removido de um único indivíduo ou organização e distribuído em blocos. Alguns exemplos de *frameworks* que utilizam desse tipo de rede são *Hyperledger* e *Corda* (DATTANI; SHETH, 2019).

Na Figura 2 pode ser visto uma representação gráfica dos tipos de rede *Blockchain*, em que a direita pode ser observado as redes que dependem de permissão de acesso e a esquerda as redes públicas. Já na intersecção entre elas é possível observar uma rede híbrida que possui certa parte aberta e outra parte de acesso privado .

Figura 2 – Distribuição dos tipos de rede *Blockchain*



Fonte: Kathleen E. Wegrzyn, Eugenia Wang, 2021

Um dos exemplos de utilização das redes públicas da *Blockchain* são as criptomoedas, já que com a maioria delas é possível realizar operações na rede de forma deliberada, ou seja, quando a rede *Blockchain* não necessita de permissão para acesso ou transferências. Já nos modelos privados podem ser definidas aplicações de utilização pessoal, assim como uma aplicação para IoT (*Internet of Things*) baseada em uma infraestrutura de rede sem fio[2] , de forma semelhante ao estilo de federação ou consórcio, como é o caso do *framework Hyperledger*, utilizado na aplicação proposta nesse trabalho.

2.1.2 Algoritmo de consenso

Algoritmos de consenso foram desenvolvidos antes do conceito de *Blockchain*. São algoritmos utilizados para tomar uma decisão para um grupo, ou seja, cada individuo do grupo apoia certa decisão e aquela com maior votos é eleita. Na *Blockchain* este algoritmo deve gerir a iteração de milhares de votos, além de entrar em um consenso que beneficie a maioria dos votantes (neste caso o beneficio é o bloco selecionado para dar continuidade a rede).

Este é o algoritmo é utilizado para definir qual será o próximo bloco gravado em uma rede *Blockchain*, sendo necessário definir algum elemento como verificador de blocos. Este elemento verificador pode ser definido de várias maneiras, porém sempre utilizando algum algoritmo, como *proof-of-work* e *proof-of-stake* por serem os mais conhecidos.

No caso de algoritmos *proof-of-work* este validador é um *hash* aleatório no qual o primeiro bloco a ser gerado com este valor será o bloco válido, enquanto os outros serão descartados. Este tipo de algoritmo de consenso é muito utilizado, e os *frameworks* mais famosos que o utilizam são o *Bitcoin* e o *Ethereum*.

Outro algoritmo de consenso é o *proof-of-stake*, que ao invés de um *hash* gerado utilizando recursos computacionais, utiliza da proporção de participação monetária da rede como

[2] Um exemplo é a proposta de cidades inteligentes, que utilizam da comunicação de pequenos dispositivos nos quais nem sempre é possível validar fisicamente qual usuário solicita o acesso

chance de escolha, ou seja, quanto maior sua posse de criptoativos maiores as suas chances de definir o próximo bloco validadoSiim (2017).

Outros algoritmos de consenso estão disponíveis, assim como algoritmos de tolerância de falha bizantina prática (pBFT), acordo bizantino federado (FBA), dentre outros. Uma *Blockchain* não está limitada a usar apenas um algoritmo de consenso. O *framework Hyperledger* tem a capacidade de utilizar diferentes algoritmos de consenso, e mais de um ao mesmo tempo, porém geralmente é implementado utilizando um algoritmo relacionado com a aplicação proposta.

2.1.3 *Smart Contract*

Ainda dentro do ramo da *Blockchain* adentramos ao escopo dos *Smart Contracts*, também conhecidos por contratos inteligentes, vem a ser definidos como:

> "Um contrato inteligente é um agente inteligente. Em outras palavras, é um programa de computador capaz de tomar decisões quando certas pré-condições são atendidas. A inteligência de um agente depende da complexidade de uma transação para o qual está programado. Os contratos podem ser transações muito simples executadas em segundos e minutos ou transações relativamente complexas e demoradas que envolvem negociações e dezenas de páginas de texto escrito com direitos e obrigações específicos que podem levar horas ou meses para ser concluído."Kolvart, Poola e Rull (2016)(Tradução própria)

Smart Contracts são amplamente utilizados para realizar operações na rede *Blockchain*, focados em operações automatizadas e de forma descentralizada. Um *smart contract* pode ser uma aplicação muito simples ou muito complexa de acordo com suas ações para execução. Um exemplo simples seria um contrato inteligente que confirma uma transação dentro da rede *Blockchain* ao realizar um pagamento. Já um exemplo mais complexo seria um algoritmo para negociação nos setores financeiros, executado em redes de criptomoedas, que podem utilizar os *smart contracts* aplicando uma matemática complexa que realiza as operações financeiras de compra e venda conforme uma previsão de lucros calculada pelo mesmo.

2.2 *FRAMEWORKS* DA REDE *BLOCKCHAIN*

A rede *Blockchain* possui diversos *frameworks*, cada qual com seus propósitos. Alguns exemplos são *frameworks* para desenvolvimento de aplicativos IoT, assim como para controle de veículos autônomos e até para propósitos gerais.

Uma boa definição dos *frameworks* da rede *Blockchain* é dada por Quasim *et al.* (2020) "Os frameworks da rede Blockchain podem ser definidas como soluções de software que simplificam o desenvolvimento e implantação de aplicativos Blockchain com um pouco de customização.", isso implica que os *frameworks* podem sem amplamente desenvolvidos e ainda mais amplamente utilizados para qualquer propósito e por qualquer pessoa com conhecimentos

básicos em programação.

Dentro da rede *Blockchain* os *frameworks* são responsáveis pela estrutura de bibliotecas utilizadas para o desenvolvimento de um aplicativo. Um aplicativo implementado na rede depende de uma infraestrutura e esta já tem o seu funcionamento derivado de nós que podem ser tanto máquinas virtuais ou físicas, e até *containers*.

Uma boa aplicação *Blockchain* deve conter um conjunto de infraestrutura, aplicativo cliente e a capacidade de ser testado em pequenas redes que não sejam as redes públicas destinadas aos usuários finais. Uma das grandes vantagens dessas aplicações é sua escalabilidade, desde que o aplicativo funcione em uma pequena rede de testes, certamente, funcionará em sua rede final e mais complexa.

Um exemplo prático do uso dos *frameworks* é o *Ethereum*, conhecido por ser um dos grandes *frameworks* da rede *Blockchain*, e também é classificado como *open source*. E o principal objetivo do *Ethereum* é ser uma máquina de usos gerais como é definido por Antonopoulos e Wood (2018):

> "[...] o Ethereum foi projetado para ser um *Blockchain* programável de propósito geral que executa uma máquina virtual capaz de executar código de complexidade arbitrária e ilimitada. Enquanto a linguagem *Script* do Bitcoin é, intencionalmente, restrita à avaliação simples de verdadeiro / falso das condições de gasto, a linguagem do Ethereum é Turing completa , o que significa que o Ethereum pode funcionar diretamente como um computador de uso geral."(ANTONOPOULOS; WOOD, 2018) (Tradução Própria).

Ou seja, mesmo o *Ethereum* sendo famoso pela sua criptomoeda ela é um subitem do objetivo principal da rede. Isso implica que basicamente qualquer aplicação pode ser desenvolvida com um *framework* da rede *Blockchain*, porém certas aplicações que requerem imutabilidade dos dados ou garantias ao executar ações, tendem a ser mais propícias nesse tipo de rede.

2.2.1 *Bitcoin*

O Bitcoin é a mais antiga moeda digital. Mas Bitcoin não é somente uma criptomoeda. Ele também dá nome ao *framework* da rede *Blockchain* que possibilita o funcionamento dessa criptomoeda. Como primeira rede Blockchain famosa, sua criação só foi possibilitada em meados de 2008, após mais de duas décadas de pesquisa e desenvolvimento por pesquisadores praticamente anônimos. Baset *et al.* (2018) relata a importância da criação do Bitcoin: "Blockchains surgiram com o Bitcoin (http://bitcoin.org/) e são amplamente considerados como uma tecnologia promissora para administrar intercâmbios confiáveis no mundo digital".

Utilizando uma rede *peer-to-peer* (P2P) para se comunicar, cada usuário da rede possui uma cópia atual, no qual todos os nós estão sendo atualizados em conjunto a cada alteração. Aliado a criptografia moderna essa rede descentralizada se tornou um campo importante na ciência da computação moderna, tendo em vista que possibilita transferências de unidades monetárias evitando o problema do "gasto duplo"(Gasto duplo pode ser definido como a capacidade de uti-

lizar o mesmo saldo para realizar duas operações financeiras diferentes, como é explicado por Ulrich (2014)) e também retirando a necessidade de um terceiro intermediário de confiança.

Cada bloco da rede Bitcoin deve ser minerado, ou seja, utilizando sub-contratos baseados em prof-of-work(PoW), as moedas da rede são calculadas garantindo a independência da política monetária e realizando o processamentos das transferências. O desenvolvimento para a plataforma do Bitcoin é baseado em protocolos alternativos, e algumas melhorias criadas pela comunidade propuseram um novo protocolo para melhorar a estabilidade e segurança da rede, outros que sugeriram alterar a proof-of-work para proof-of-stake (PoS) em sub-redes do Bitcoin, também para gerar fatores de dupla autenticação, dentre outros.

2.2.2 *Ethereum*

Junto do *Bitcoin* na área das criptomoedas mais famosas da *Blockchain* o *Ethereum*, como definido anteriormente, tem o objetivo de que suas aplicações possam ser utilizados em qualquer tarefa que uma máquina normal poderia utilizá-la.

O *Ethereum* possui como componentes a sua rede no estilo P2P, suas regras de consenso, transações, a sua máquina de estados, o algoritmo utilizado para consenso, as estruturas de dados, os clientes e a segurança econômica Antonopoulos e Wood (2018). A segurança atual do *Ethereum* é gerada por PoW utilizando o algorítimo conhecido por *Etash*, o qual é o algorítimo utilizado para mineração de *Ethereum* atualmente, porém em breve a rede sofrerá uma atualização e seu funcionamento será dado a partir de PoS. Ainda em termos de segurança o algoritmo de consenso da rede atual utiliza o modelo chamado por *Nakamoto Consensus*, semelhante ao modelo de consenso do *Bitcoin*, porém como mencionado ao mudar a forma de processamento do consenso para PoS o sistema será migrado para uma metodologia de votação ponderada.

Os nós dessa rede armazenam localmente sua estrutura de dados assim como um banco de dados, dentro deles são armazenadas as transações da rede e utilizando uma estrutura de *hash* serializada é armazenado o estado do sistema. A máquina de estados do *Ethereum* conhecida por *Ethereum Virtual Machine* (EVM) processa as transações da rede que são armazenadas em uma pilha executada em *bytecode*. Para se gerar um programa a ser executado no EVM é utilizado uma linguagem de programação chamada *Solidy*, que após compilado o algoritmo é obtido um programa em *bytecode* que pode operar na EVM, este programa é chamado de *Smart contract* na rede *Ethereum*.

As regras de consenso da rede estão disponíveis em um artigo escrito por Wood *et al.* (2014) nomeado *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*, este artigo é reconhecido por *yellow paper* do Ethereum e contém a estruturação dos protocolos de consenso da rede.

2.2.3 *Hyperledger Fabric*

Dentro deste trabalho será dado enfoque no *framework* da rede *Hyperledger Frabric*. Este *framework* é derivado da Fundação *Hyperledger* que se auto define como:

> "A *Hyperledger Foundation* é uma organização sem fins lucrativos que reúne todos os recursos e infraestrutura necessários para garantir ecossistemas prósperos e estáveis em torno de projetos de Blockchain de software de código aberto."(HYPERLEDGER, 2020)(Tradução própria)

Esta Fundação mantém hospedados muitos projetos, inclusive aqueles de nível empresarial. Um dos projetos consolidados é o *Hyperledger Fabric*, um dos *frameworks* descritos neste trabalho, utilizado para desenvolver a aplicação da rede *Blockchain*.

Entre as principais diferenças do *Hyperledger Fabric* para o *Ethereum*, incluem que *Hyperledger Fabric* não trabalha com a *Blockchain* de conceito pública, ele trabalha com a rede no formato privado e com permissão Hyperledger (2020). Neste formato de rede não é necessário registros de prova de trabalho, pois não é permitido que usuários desconhecidos acessem a rede. Ao invés da prova de trabalho, é utilizado um provedor de serviços de associação (MSP) confiável para que possa ser inseridos os membros na rede.

Outros atributos disponíveis em uma rede utilizando *Hyperledger Fabric* é a capacidade de utilizar aplicações "*plugins*"que podem ser inseridas e removidas da rede, assim como mecanismos de consenso e diferentes MSP's são suportados pela rede. Uma outra capacidade dessa rede é a possibilidade de se criar canais exclusivos entre grupos de clientes utilizando um livro razão separado do principal e exclusivo para aquele grupo: "Se dois participantes formarem um canal, esses participantes - e nenhum outro - terão cópias do livro-razão desse canal"(HYPERLEDGER, 2020).

Alguns pontos importantes a serem considerados em uma rede utilizando *Hyperledger Fabric* são seus *Smart Contracts*, a privacidade da rede, os métodos de consenso e a forma como o livro razão é armazenado. O consenso neste tipo de rede é derivado de várias formas, mas possui um conceito básico que é explicado da seguinte forma: "As transações devem ser gravadas no livro razão na ordem em que ocorrem, mesmo que possam ser entre diferentes conjuntos de participantes na rede."Hyperledger (2020). As transações além disso, ainda precisam possuir um método implementado que rejeite as transações ruins que são inseridas maliciosamente ou erroneamente. Mesmo assim a rede do *Hyperledger* foi desenvolvida de forma que aqueles que estão iniciando nesta rede têm a opção de escolher o melhor formato de consenso para sua aplicação de acordo com suas necessidades.

Assim como no consenso a privacidade dessa rede também depende das necessidades dos usuários. Oferecendo assim um suporte a redes corporativamente abertas ou também no qual a privacidade é um requisito operacional. Cada usuário utilizando a rede possui uma cópia do livro razão de cada rede em que o mesmo está inserido. O funcionamento do livro razão dentro do *Hyperledger Fabric* funciona com um método chamado razão compartilhada que possui dois

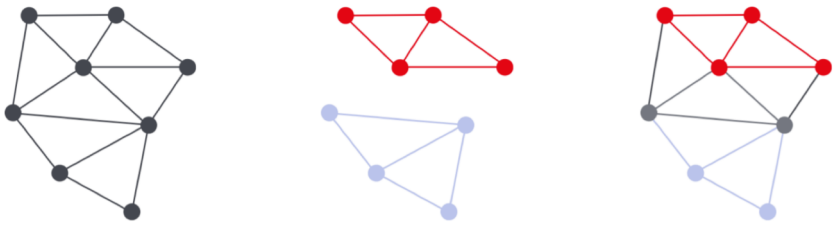
componentes que combinados garantem a legitimidade da rede. Esses componentes são chamados de Estado global e *log* de transações. Eles podem ser definidos como o banco de dados da rede e o histórico de atualizações respectivamente. Os *Smart contracts* utilizados são escritos utilizando *chaincode*, normalmente possuem sua iteração apenas com o componente chamado de estado global. Quando é necessário realizar alguma alteração na *Blockchain*, esse aplicativo é chamado por um agente fora da mesma. Para desenvolver os aplicativos em *chaincode* são recomendadas linguagens como *Go*, *Java chaincode* e *Node.js*.

2.2.4 *Corda*

Semelhante ao *Hyperledger Fabric* também possui uma rede *Blockchain* privada com consenso, e assim como o *Ethereum* utiliza uma máquina virtual com grande poder de processamento para executar seus códigos. E ainda, comparado ao *Bitcoin*, possui semelhanças como os estados imutáveis que são criados e consumidos pelas transações, essas ainda podem ter múltiplas entradas e saídas com os contratos criados sempre tendo o mesmo resultado, pois não possuem a capacidade de interagir ou armazenar algo Brown *et al.* (2016).

A especialização desse *framework* é para utilização em instituições financeiras regulamentadas, sua inspiração foi em sistemas *Blockchain*, porém com um design mais adequado ao seu propósito. Outro ponto apontando por Brown (2018), é que a rede *Corda* possui a promessa de poder operar com aplicativos em um contexto empresarial ou público, sem possuir as desvantagens da rede pública e também sem as limitações das redes privadas que não possuem comunicações externas.

Figura 3 – Exemplo de redes públicas, redes privadas não interoperáveis e a rede Corda respectivamente



Fonte: Adaptado de Brown (2018).

Na Figura 3 é mostrado da esquerda para direita um exemplo de rede pública, um exemplo de rede privada e pública separados e um exemplo da rede que a *Corda* utiliza. Esta rede possui a vantagem de interligar uma rede *Blockchain* pública com uma rede privada. Esta ligação traz a possibilidade de armazenar dados privados e caso seja necessário é possível compartilhar estes dados sem gerar outra rede específica para isso.

2.2.5 *Cardano*

A princípio um dos primeiros *frameworks* a implementar o serviço de PoS como protocolo verificador. O protocolo utilizado é chamado de *Ouroboros*, no qual os blocos são gerados selecionando aleatoriamente um líder de seção que produzirá o novo bloco. O líder é escolhido aleatoriamente, porém as chances de ser escolhido são proporcionais a quantidade de criptoativos que ele detém.

A Cardano é uma rede pública com foco no desenvolvimento baseado em evidências e seu foco também abrange sustentabilidade, escalabilidade e transparência como é descrito por Kammerer (2022). Seu desenvolvimento vem sendo focado em abranger uma alta gama de casos de uso, assim englobando soluções que tentam resolver problemas em vários setores da indústria. Os focos atuais em desenvolvimento envolvem o campo médico, campo de finanças, sistema de verificação de credenciais para o governo, certificação e rastreabilidade de produtos agrícolas, auditoria de produtos para evitar a falsificação e sistema de armazenamento de credenciais em *Blockchain*.

Este *framework* também é o responsável por criar e manter a criptomoeda ADA, que é utilizada para processar as transações efetuadas em sua rede. Seus *smart contracts* desenvolvidos na linguagem *Pluto* e assim como no *Ethereum*, esta linguagem também é Turing completa. Os contratos inteligentes deste *framework* ainda não estão completamente operacionais. Atualmente só estão disponíveis contratos financeiros desenvolvidos em *marlowe* em sua rede de produção, os contratos inteligentes desenvolvidos em *Plutus* não estão integrados a rede oficial.

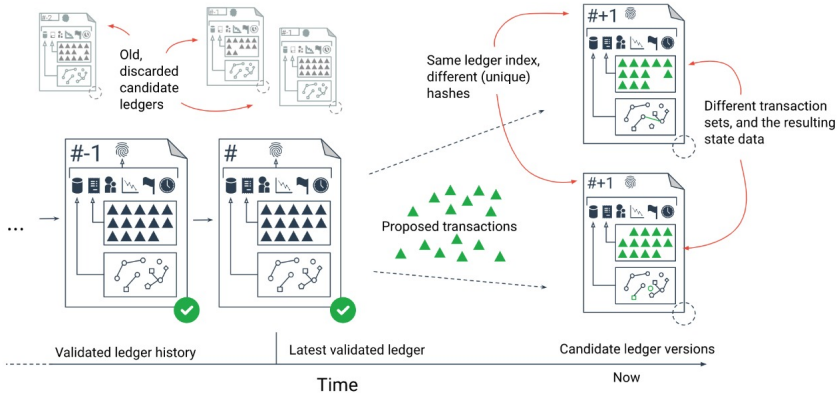
2.2.6 *Ripple*

Este *framework* é focado em transações financeiras e soluções "cripto embarcadas" para empresas, a *Ripple* tem seu foco em possuir transações mais rápidas, transparentes e econômicas do que instituições financeiras padrões, como é dito por Schwartz *et al.* (2022). Sua rede é do tipo pública e também é pertencente ao criador e administrador do criptoativo XRP.

Seu tipo de algoritmo de consenso é um pouco diferente, pois utiliza um sistema de validação próprio que possui várias versões do livro razão candidatas contendo o mesmo índice, porém apenas uma dessas versões se torna a oficial com aquelas transações gravadas no histórico da rede. Este processo de consenso é chamado de *Ripple Protocol Consensus Algorithm* (RPCA), ou seja, algoritmo de protocolo de consentimento *Ripple* desenvolvido por Schwartz *et al.* (2014).

O funcionamento do RPCA está ilustrado na Figura 4, no qual várias propostas de transações são recebidas pelo último bloco válido nesta rede, e estas transações são distribuídas em novos blocos candidatos com mesmo índice, porém com *hashs* diferentes. Após recebidos estes blocos candidatos um deles é escolhido para ser o sucessor da rede e então todos os outros são descartados. Somente as transações contidas naquele bloco são armazenadas na rede, de forma

Figura 4 – Funcionamento da seleção de bloco na rede utilizando *Ripple*



Fonte: Adaptado de Schwartz *et al.* (2022).

que para uma transação seja efetuada ela deve estar contida em um bloco escolhido da rede.

3 MATERIAIS E MÉTODO

Aqui é apresentado os métodos a serem utilizados de forma que fique claro como foi o desenvolvimento da aplicação. São descritos os principais métodos que foram utilizados para a produção do manual de execução da aplicação utilizando *Blockchain* implementando *Hyperledger Fabric*. Os métodos para a produção do comparativo entre os *frameworks* também foram apresentados neste capítulo. É listado do mesmo modo os materiais utilizados para que os objetivos propostos fossem alcançados.

3.1 MATERIAIS

Os principais itens utilizados foram os materiais de pesquisa e um computador com conexão à Internet. O computador utilizado possui um processador Intel I7-6700HQ, 24Gb de memória RAM, placa de vídeo dedicada GTX 970M com 3Gb de VRAM e SSD NVME de 1Tb de armazenamento, utilizando o sistema operacional Linux Mint 20.10 LTS cinnamon. Os softwares utilizados foram, terminal Linux para execução de *scripts* e comandos *shell*, *VS Code* para edição de códigos, *Curl, Git, Docker, GoLang, NodeJs e NPM* para compilar e executar a aplicação. Dentre os materiais de pesquisa constam artigos publicados relacionados a *Blockchain* e livros sobre o assunto, documentação sobre as linguagens de programação utilizadas, documentação sobre os *frameworks* e documentação sobre bibliotecas para programação.

3.2 MÉTODO

Como, no caso deste trabalho, mais de um objetivo foi definido, devido aos métodos abordados não é possível utilizar do mesmo para ambos. Portanto é visado trazer uma abordagem para cada item, que em conjunto complementem totalmente o que foi proposto.

3.2.1 Análise dos *frameworks*

A primeira parte deste trabalho teve como foco o estudo dos principais *frameworks* disponíveis, coletando informações sobre seu funcionamento, características e utilização, destacando também possíveis desvantagens. o estudo em geral não propõe definir qual é o melhor *framework* da *Blockchain*, mas sim trazer um comparativo entre cinco das redes famosas para que se possa ter um entendimento mais abrangente de qual *framework* pode ser mais indicado dependendo do caso de uso.

Uma tabela contendo o comparativo entre cinco *frameworks* foi compilada com as informações obtidas através da pesquisa, esta tabela contém dados sobre os recursos de privacidade dos *frameworks*, processos de segurança, custos e velocidade de transações, sua abordagem com

contratos inteligentes, tipo de plataforma utilizada e sua capacidade de escalabilidade. Também foram compiladas outras duas tabelas uma com as vantagens e outra com as desvantagens mais impactantes em cada *framework*, nos quais não possuem a mesma métrica de comparação pois cada *framework* dispõe de suas particularidades.

3.2.2 Manual de execução da aplicação

O manual de execução da aplicação foi desenvolvido em conjunto ao desenvolvimento, de forma que não gerasse duplicidade de informações. Outro ponto para seu desenvolvimento ser integrado é a maior facilidade de entender o passo a passo acompanhado do código criado, desta forma a visualização do conteúdo fica mais iterativa com o leitor.

3.2.3 Desenvolvimento da aplicação

Foi proposta uma aplicação de uso geral, em que deve-se ter a possibilidade de atribuir produtos a pessoas. Esta aplicação, de um modo geral, tem um formato mais simples de forma que o desenvolvimento fosse simplificado e fácil de ser associado pelo leitor. Para o desenvolvimento foi utilizado um código base fornecido pela biblioteca *Goledger*, que simplifica as funções de criação de redes e organizações podendo trazer o foco somente a área de programação para *Blockchain*. Esta biblioteca faz parte do *framework Hyperledger* que foi selecionado para que a aplicação fosse desenvolvida. A seleção do *framework* se deu após a análise dos mesmos. Em que o *Hyperledger* foi o *framework* mais indicado para a aplicação proposta.

4 ANÁLISE DE *FRAMEWORKS*

Dentro da área de comparativos de *frameworks* existem trabalhos publicados sobre o tema, apontando as diferenças dos itens que compõem os *frameworks* assim como é mostrado na Tabela 1.

Tabela 1 – Tabela contendo algumas relações entre os *frameworks Ethereum*, *Hyperledger Fabric* e Corda

Características	**Ethereum**	**Hyperledger**	**Corda**
Descrição da plataforma	Plataforma genérica da Blockchain	Plataforma Modular da Blockchain	Plataforma de razão distribuída especializada na indústria financeira
Governança	Desenvolvedores do Ethereum	Fundação Linux	R3
Modo de operação	Sem permissividade, pública ou privada	Com permissividade, Privada	Com permissividade, Privada
Consenso	Mineração baseada em PoW, Nível de razão	Ampla variedade de consensos que permite múltiplas aproximações, Nível de transação	Compreensão específica de consensos, Nível de transação
Smart Contracts	Código de Smart Contract (ex: Solidy)	Código de Smart Contract (ex: Go, Java)	Código de Smart Contract (ex: Kotlin, Java), Smart legal Contract (para propósitos legais)
Moeda	Ether, Tokens via Smart Contract	Nenhuma, Moedas e tokens via chaincode	Nenhuma

Fonte: Adaptado de Valenta e Sandner (2017)

Um outro exemplo desses trabalhos é uma analise entre os *frameworks Ethereum*, *Hyperledger Fabric* e Bitcoin para desenvolvimento de um aplicativo na área da saúde. Esse estudo foi desenvolvido por Agbo e Mahmoud (2019) e chega a conclusão que o *Hyperledger Fabric* possui as melhores condições para o desenvolvimento dessa aplicação, já que contém os recursos mais completos para este quesito. O artigo cita também que é interessante desenvolver uma aplicação nessa área para verificar fatores como escalabilidade, latência e taxas de transferências do *framework* utilizado.

Um comparativo básico foi feito entre 5 dos *frameworks* famosos da rede *Blockchain*, excluindo o Bitcoin já que ele não possui a possibilidade de executar contratos inteligentes. Está analise visa aprofundar o conhecimento sobre o foco geral de cada um dos *frameworks*, buscando enquadrar a aplicação no qual seja o *framework* mais indicado.

Foram abordados os recursos de privacidade de cada *framework*, a segurança aos dados salvos, os custos por transação, sua capacidade utilizando contratos inteligentes, a descrição de utilização da plataforma, a velocidade entre as transações e a escalabilidade da rede. Estes pontos são abordados da seguinte forma:

- Recursos de privacidade: elementos que validam as transações, e provas de validação independente de outros pares;

- Segurança: possibilidade de acesso aos dados compartilhados;
- Custo por transação: Custo da transação efetuada, caso dependa de algum criptoativo associado a moeda o custo será mais elevado. Caso a transação dependa apenas da rede o custo será mais baixo;
- Contratos inteligentes: capacidade do *framework* em utilizar contratos inteligentes para diferentes aplicações;
- Descrição da plataforma: descrição de modo de utilização da plataforma, caso seja genérica, modular ou específica para um elemento;
- Velocidade de transações: Capacidade do *framework* em transações por segundo. Teoricamente quanto maior a capacidade da rede neste elemento mais transações esta rede pode realizar em um dado espaço de tempo;
- Escalabilidade: capacidade de inserir e remover funções de um *framework* de forma simples e sem afetar o resto da execução. Será avaliado em baixa, média e alta, quanto maior a facilidade de inserir e remover elementos maior será a escalabilidade.

Utilizando como base algumas tabelas encontradas nos estudos fornecidos por Dattani e Sheth (2019), Agbo e Mahmoud (2019), Benji e Sindhu (2019), Garriga *et al.* (2021), Clincy e Shahriar (2019), Valenta e Sandner (2017) e Sajana, Sindhu e Sethumadhavan (2018), compilando as tabelas em conjunto do conhecimento abordado por Pelt (2019), Cardano (2021), Nadir (2019), Nakamoto (2008) e Wood *et al.* (2014), foi possível concentrar várias informações sobre o tema em uma tabela resumida focando nos *frameworks Ethereum*, *Hyperledger Fabric*, *Corda*, *Cardano* e *Ripple*. Estes dados acumulados estão disponíveis na Tabela 2.

Foram selecionados somente estes cinco *frameworks* pois possuem aplicação com grande variabilidade e também são alguns dos mais famosos empregados atualmente. Entretanto é notável que existem uma infinidade de outros *frameworks*, assim como *EOS*, *IOTA*, *WAVES*, *Quorum*, dentre outras que poderiam estar associados entre os principais aqui dispostos.

Incluindo também o que foi escrito por Quasim *et al.* (2020) é possível definir algumas vantagens e desvantagens para cada *framework* abordado. Suas vantagens estão representadas na Tabela 3 e as desvantagens na Tabela 4. Utilizando destas tabelas como base é possível definir qual *framework* mais indicado para a aplicação em específico.

Tabela 2 – Comparações entre os *frameworks*

Elemento	Ethereum	Hyperledger	Corda	Cardano	Ripple
Recursos de privacidade	Transações privadas e provas de zero conhecimento experimentais	Canal privado, transações privadas e prova de zero conhecimento	Canal privado, transações privadas e prova de zero conhecimento	Validações geradas por scripts com retorno booleano (true ou false)	Nó que valida e confirma as transações
Segurança	Acesso público aos dados, porém suporta redes com permissões	Rede de consórcio, necessita de permissão para acesso aos dados	Rede de consórcio, necessita de permissão para acesso aos dados	Rede pública, de código aberto	Rede privada, necessita de permissão para acesso aos dados
Custo por transação	Alto, porém varia conforme o custo do criptoativo	Baixo, somente custos de manutenção de infraestrutura Pode conter criptoativo incorporado em sua rede, neste caso os custos estariam vinculados ao criptoativo	Baixo, somente custos de manutenção de infraestrutura	Médio, variável conforme o custo do criptoativo	Médio, variável conforme o custo do criptoativo
Contratos inteligentes	Sim, produzido em *Solidy*	Sim, produzido em Go e Java	Sim, produzido em Java Contrato inteligente judicialmente vinculado	Sim, produzido em *Plutus*, *Marlowe* e *Glow*	Sim, produzido utilizando XRP Ledger com escrow
Descrição da plataforma	Plataforma genérica *Blockchain*	Plataforma *Blockchain* modular	Plataforma *Blockchain* especializada na indústria financeira	Plataforma *Blockchain* modular	Plataforma concentrada em transações de criptoativos
Velocidade de transações	15 transações por segundo	2.000 transações por segundo	200 transações por segundo	1.000 a 1.000.000 de transações por segundo, aumenta conforme o número de pares conectados	1500 transações por segundo
Escalabilidade	Baixa	Alta	Alta	Média	Baixa

Fonte: Autoria própria

Tabela 3 – Vantagens dos *frameworks*

Ethereum	Hyperledger	Corda	Cardano	Ripple
Alta capitalização	Arquitetura modular	Sem custos de transação relacionado a criptoativos	Alta escalabilidade	baixa taxa de transação com custo de 0.00001 XRP
Time de desenvolvimento forte	Consultas semelhantes a SQL	Especializado na indústria financeira	Ótima equipe de desenvolvimento	Possibilidade de cancelar transações
Confiabilidade comprovada	Modelo de rede permissivo	Modelo de rede permissivo	Plataforma estruturada em camadas	Suporte a grandes bancos
Permite alavancamento em Blockchain	Performance e escalabilidade optimizada	Construído na ferramenta padrão da indústria R3	Manutenção facilitada	Modelo de rede permissivo
Alto tráfego diário	Sem custos de transação relacionado a criptoativos	Tem capacidade de gerar consenso para contratos individuais	Taxas de transações por segundo muito elevadas	Boa taxa de transação por segundo

Fonte: Autoria própria

Tabela 4 – Desvantagens dos *frameworks*

Ethereum	Hyperledger	Corda	Cardano	Ripple
Taxa de transferências por segundo muito baixa	Arquitetura do Fabric muito complexa	Estrutura customizada para o setor financeiro	Taxa de transferência utilizando criptoativos	Baixa escalabilidade
Framework centralizado que a opinião dos desenvolvedores é mais importante do que a da comunidade	Casos de uso específico pouco abordados	A utilização de pessoas para confirmar autenticidade dos documentos reduz a legitimidade das verificações	Contratos inteligentes não estão amplamente disponíveis	Equipe de gerência pode bloquear os criptoativos dos usuários
Alto custo de transações	Não possui integração entre rede privada e pública	Uso limitado ao setor financeiro		Arquitetura complexa
Atualização para PoS com problemas	Implementação requer um pouco mais de experiência dos programadores	Capacidade de transações por segundo baixa		Ripple Labs detém 65% dos criptoativos da plataforma

Fonte: Autoria própria

Neste trabalho será desenvolvido uma aplicação com foco básico em transmissão e associação de itens a pessoas, portanto sua capacidade de transações em geral deve ser alta. Além disso, os produtos devem ser genéricos não apenas ativos financeiros. Por motivos de utilização geral, o *framework* escolhido também deve ser de código aberto e possuir a capacidade de inserção de módulos variados de acordo com as necessidades da aplicação.

Por estes motivos, além deste *framework* já possuir uma base sólida e utilizada em aplicações de produção foi escolhido o *Hyperledger Fabric* como base para o desenvolvimento. Outro *framework* possível a ser selecionado é o Cardano, porém suas bases para contratos inteligentes não estão totalmente desenvolvidas. Portanto em um futuro em que já esteja melhor desenvolvido esta base seria interessante realizar uma produção de aplicação com este *framework*.

A aplicação será desenvolvida de maneira simplificada para facilitar o entendimento dos passos a serem executados de forma que a plataforma opere sem complicações devido a erros de implementação do código. Previamente ao desenvolvimento da aplicação, será abordado uma *demo* fornecida pela biblioteca *CC-Tools*, que corresponde a um aplicativo desenvolvido para *Blockchain* no formato de biblioteca, no qual é possível inserir livros e pessoas, além de poder associar estes livros a pessoas e a bibliotecas.

5 DESENVOLVIMENTO E EXECUÇÃO DE UMA APLICAÇÃO *BLOCKCHAIN*

Este capítulo é dedicado a apresentar um passo a passo de como desenvolver uma aplicação funcional na rede *Blockchain* que pode ser implementado tanto numa rede de testes local quanto em uma rede de produção online. A área de desenvolvimento em *Hyperledger* conta com livros como *Hands-On Smart Contract Development with Hyperledger Fabric V2* escrito por Zand, Wu e Morris (2021) que compõem do desenvolvimento base em uma plataforma utilizando o *framework Hyperledger*. Além de contar com artigos como o escrito por Yamashita *et al.* (2019), que trabalha com os potenciais riscos associados aos contratos inteligentes de uma aplicação desenvolvida utilizando deste mesmo *framework*.

Estes trabalhos contém informações relevantes ao desenvolvimento de aplicações utilizando *Hyperledger*, porém são voltados a um público que já possua pelo menos um conhecimento básico sobre *Blockchain*. A aplicação aqui desenvolvida aborda uma área inicial de desenvolvimento, na qual o público abordado não tenha experiência com programação para *Blockchain*. O material base para esse tutorial (códigos e *scripts*) foi elaborado pela plataforma *GoLedger*[1] que visa simplificar a maneira de produção de códigos para *Blockchain* trazendo uma biblioteca com alguns elementos básicos já implementados.

5.1 PRÉ-REQUISITOS DE INSTALAÇÃO

Alguns itens são imprescindíveis para executar esta aplicação de maneira eficiente. A partir da instalação Linux base, no qual é recomendado o sistema Ubuntu 22.04 LTS; é necessária a instalação dos itens a seguir iniciando com *git*, *docker* versão 19 ou superior, GoLang versão 1.14 ou superior, NodeJs versão 10, *Hyperledger Fabric*k versão 1.4, além de ter acesso ao GCC. Também são necessários conhecimentos básicos de programação e utilização de terminal Linux.

Após a instalação destes itens é um passo importante testar sua funcionalidade. Uma maneira efetiva é utilizando os comandos abaixo no terminal, que respectivamente mostrará a versão do *Curl*, uma execução de *Docker* que mostrará "*hello world*", a versão do *Docker-compose*, a versão do GoLang, a versão do NodeJs e também a versão do NPM.

```
curl --version
docker run hello-world
docker-compose --version
go version
node -v
npm -v
```

[1] https://goledger.com.br/

É importante que todos estes elementos estejam funcionais para que os próximos passos sejam executados perfeitamente e sem imprevistos.

5.2 INICIALIZANDO O PROJETO

O primeiro passo executado foi clonar o repositório do *github* que contém a base do projeto utilizado. Para isso é necessário empregar os comandos abaixo, a partir do diretório "*home*"no Linux, dessa forma é feita uma cópia atualizada dos arquivos necessários para se ter uma introdução a *Blockchain*. Após isso é navegado com o terminal para a pasta que acabou de ser baixada.

```
cd $HOME \
git clone https://github.com/goledgerdev/cc-tools-demo.git \
cd cc-tools-demo
```

Agora o comando a ser executado é:

```
./installPreReqUbuntu.sh
```

Desta forma, realizada a instalação dos elementos base necessários para execução do projeto. Assim que estiver concluído irá aparecer no terminal a mensagem "*Enviroment configured*".

5.3 EXECUTANDO E TESTANDO A APLICAÇÃO

Esta seção possui como foco executar o código base do aplicativo de testes e também mostrar seu funcionamento em uma rede pré-definida com três organizações atuando no mesmo *chaincode* utilizando-se de três clientes. Notando que estas são as especificações pré-definidas pelo projeto base.

5.3.1 Executando a aplicação

O primeiro passo foi acessar a pasta *fabric* utilizando o comando:

```
cd fabric
```

A seguir foram removidas as pastas, junto de seu conteúdo, em que ficam armazenados os certificados digitais conhecidos como *Membership Service Provider* (MSP). Estes certificados são aqueles responsáveis por conferir autenticação as organizações. Cada certificado é gerado por uma autoridade de certificação (CA) utilizando da plataforma *Hyperledger Fabric CA*, no nosso contexto cada uma das organizações é representada por uma CA exclusiva.

O comando utilizado para apagar as pastas e seus conteúdos é:

```
rm -rf crypto-config channel-artifacts ca
```

Após executado, foi utilizado o comando para gerar novos certificados e sair da pasta fabric:

```
./startDev.sh generate
cd ..
```

O próximo passo foi executar o download das dependências tando do GoLang quanto do NodeJs. Para isso foram executados na sequência respectivamente o comando para entrar na pasta que contém o *chaincode*, baixar suas dependências, sair desta pasta, entrar na pasta que contém o servidor REST, executar o comando para baixar suas dependências, pressionando então o comando para sair e voltar a pasta principal do projeto.

```
cd chaincode
go mod vendor
cd ..
cd rest-server
./npmInstall.sh
cd ..
```

REST é um padrão utilizado para comunicação de servidores, tanto de formas simples como complexas. Neste caso como é implementado utilizando comunicação no protocolo HTTP as iterações são efetuadas utilizando os comandos "POST", "PUT", "GET"e "DELETE". O servidor REST fornecido é utilizado para realizar a comunicação entre a *Demo* fornecida pela biblioteca do *framework*, e uma aplicação de interface *web* em que nossas organizações estarão conectadas e então farão a utilização da rede. Não é um elemento necessário para o funcionamento da rede, porém simplifica o processo de aprendizagem sobre como uma rede *Blockchain* funciona no mundo real e como pode ser aplicada.

Após os passos anteriores serem executados, o projeto está pronto para iniciar a construção da rede. Foi então necessário utilizar o comando:

```
./startDev.sh
```

Ao utilizarmos deste comando, são iniciadas as tarefas que consistem em criar os *containers* dos nós da rede *Blockchain* associados as três organizações que iremos utilizar: criar o *ledger* para registros e permissões, ingressar as organizações no *Hyperledger Frabric*, instalação da API de gerenciamento da rede *Hyperledger Fabric*, instalação do *chaincode* nos pares endossantes, instanciar o *chaincode* no canal e a implantação dos *containers* de servidores REST já possuindo a correta configuração criptográfica que representa para cada organização um *Hyperledger Fabric Client*.

5.3.2 Testando a aplicação

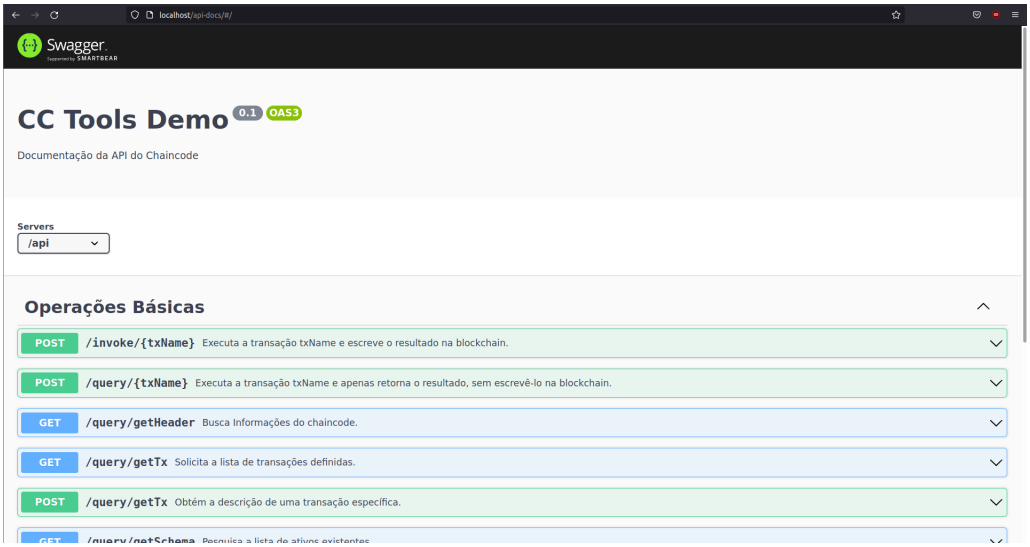
Com exito das operações anteriores agora é possível realizar testes funcionais nesta rede local. O primeiro passo é utilizar a ferramenta *cc-webclient*, já integrada ao sistema, para executar três aplicações que serão conectadas as organizações da nossa rede. Para isso utilizamos os comandos abaixo interligando estas aplicações as portas 8080, 8090 e 8100 respectivamente ao *localhost*.

```
./run-cc-web.sh 8080 &
./run-cc-web.sh 8090 &
./run-cc-web.sh 8100 &
```

As organizações utilizadas no sistema são representadas como Org1, Org2 e Org3, sendo elas as responsáveis por gerenciar os ativos da rede (Os dados, ou *assets*) e utilizar das transações (métodos para alterar os ativos que necessitam de funções no *chaincode* para operar, também chamados de *transactions*) elaboradas com suas devidas permissões. A base de cada organização, após os passos anteriores, ficaram online e podem ser verificadas em *http://localhost* nas portas 80, 980 e 1080 para as organizações Org1, Org2 e Org3 respectivamente.

Utilizando das mesmas portas de acesso e adicionando "/api-docs/" ao endereço, é possível acessar a documentação das funções disponíveis para as organizações, assim como é visto na Figura 5. Além ter a possibilidade de teste para as transações de forma básica para verificar seu funcionamento.

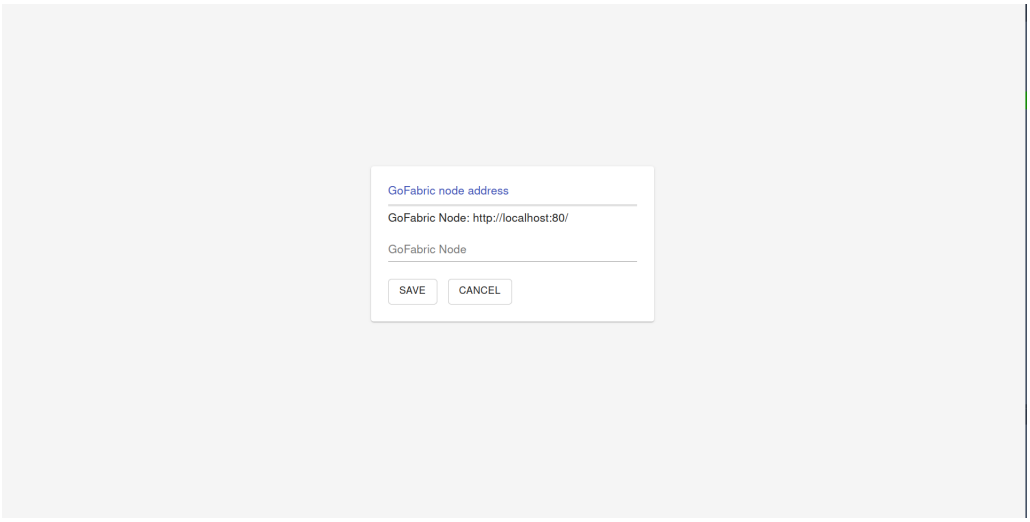
Figura 5 – Tela de API das Organizações



Fonte: Autoria própria.

Agora ao acessar as portas criadas anteriormente com *webclient* somos direcionados a uma tela, mostrada na Figura 6, para configurar o acesso do *webclient* com o Node local. Respectivamente para as organizações Org1, Org2 e Org3 devemos configurar os links de localhost com as portas 80, 980 e 1080 nos *webclients* das portas 8080, 8090 e 8100.

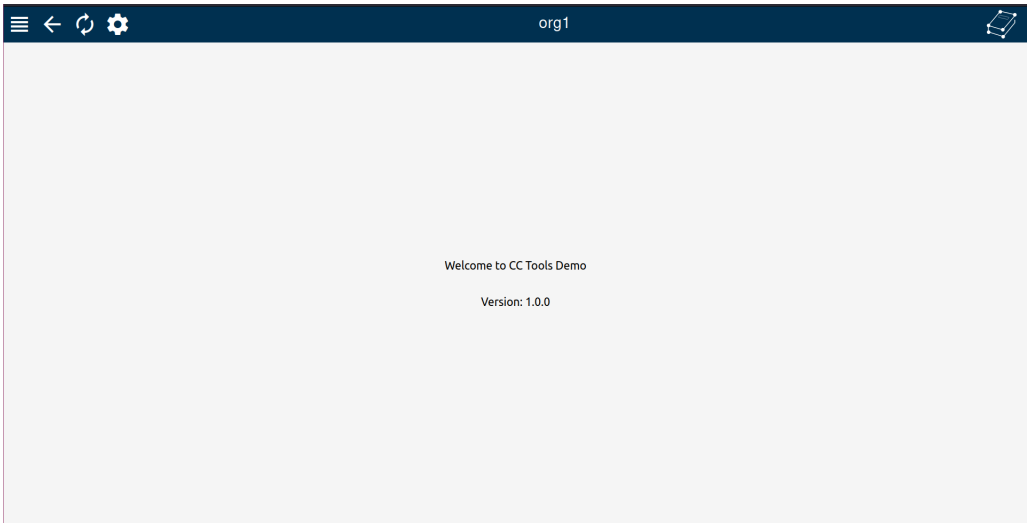
Figura 6 – Tela de configuração do servidor node



Fonte: Autoria própria.

Agora nas portas 8080, 8090 e 8100 temos disponíveis as interfaces respectivas a Org1, Org2 e Org3, é representado na Figura 7. Nessas telas é possível serem feitos cadastros de livros, cadastros de pessoas, cadastros de biblioteca, atribuição de livros a uma biblioteca, atribuir posse de livros a uma pessoa, criar uma informação secreta que só pode ser visualizada pelas organizações cadastradas, além de poder editar as informações já existentes de acordo com as permissões das organizações.

Figura 7 – Tela inicial da Organização 1



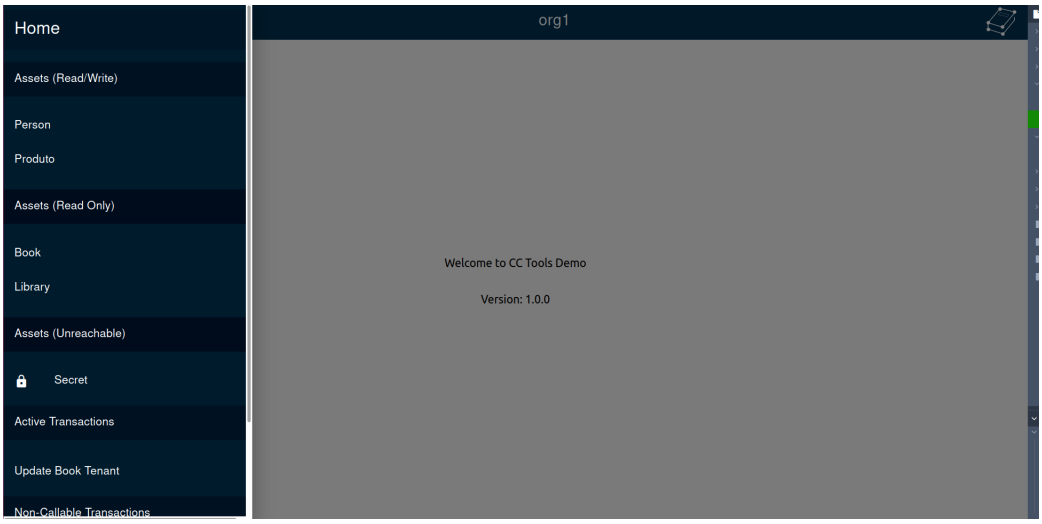
Fonte: Autoria própria.

5.3.3 Criando elementos e associando itens

Com a aplicação em funcionamento é possível agora testar sua utilização por um usuário. Cada tela que possui uma função de transferência contém também um botão "CURL"que

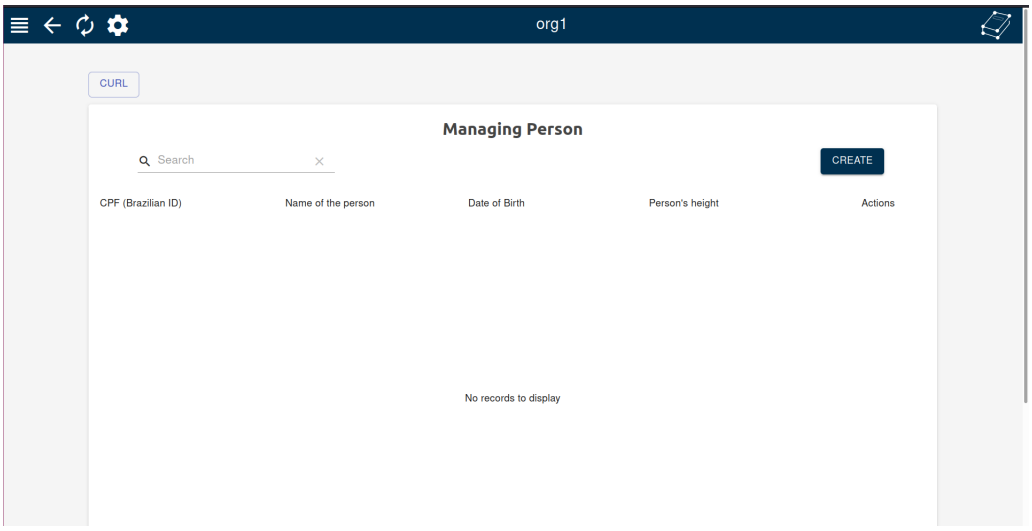
exibe a operação que irá ser realizada. Um ótimo exemplo é na Org1 na qual pode-se efetuar o cadastro de uma pessoa. Acessando o botão no canto superior esquerdo e clicando em Person, somos redirecionados a tela de pessoas cadastradas. O Menu lateral está representado na Figura 8 e a tela que exibe as Persons está representada pela Figura 9.

Figura 8 – Tela do menu lateral da Organização 1



Fonte: Autoria própria.

Figura 9 – Tela que exibe os ativos pessoa cadastrados

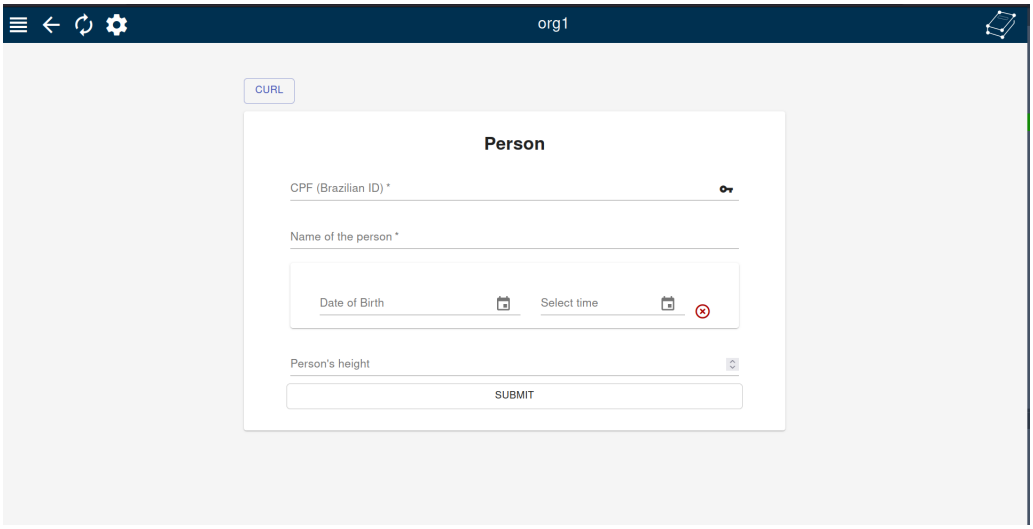


Fonte: Autoria própria.

Como a tela está vazia devemos clicar no botão CREATE para que sejamos direcionados a tela de criação do ativo pessoa, a qual é exibida na Figura 10. Esta tela conta com as informações cadastradas do ativo, na qual pode-se solicitar a exclusão do mesmo, ou também verificar seu histórico de alterações que fica armazenado no livro-razão.

Como não há nenhum ativo Pessoa cadastrado, utilizamos desta tela para realizar um cadastro. Inserindo as informações necessárias e antes de clicar em SUBMIT pode-se verificar

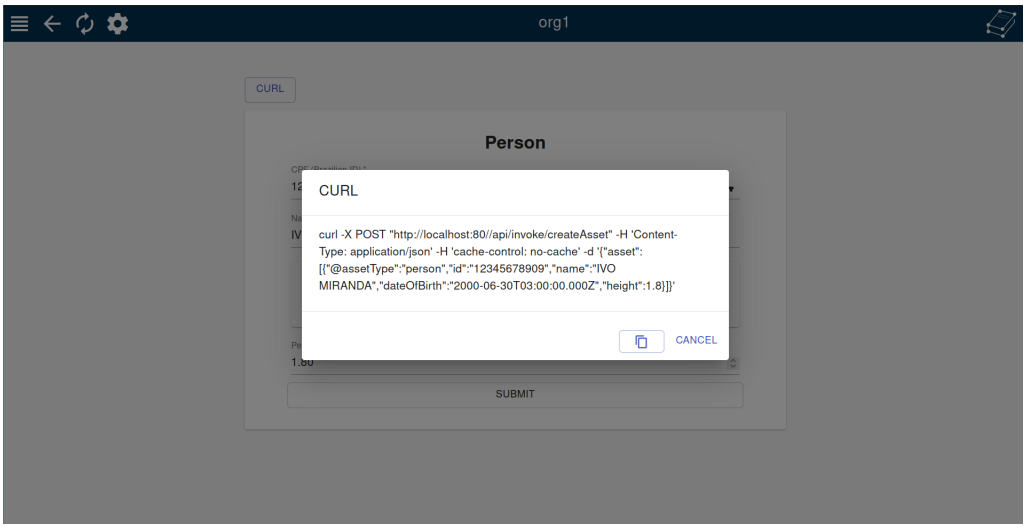
Figura 10 – Tela de cadastro de pessoa



Fonte: Autoria própria.

no botão CURL quais serão as informações transmitidas e qual transação será utilizada. É exibido na Figura 11 a transação que será efetuada possuindo os campos de cadastro já completos. Ao clicar em SUBMIT é gerada a transação, portanto nosso ativo já está armazenado em nosso livro razão, podendo consultá-lo na tela anterior a de criação de pessoas.

Figura 11 – Tela de informação da transação a ser executada



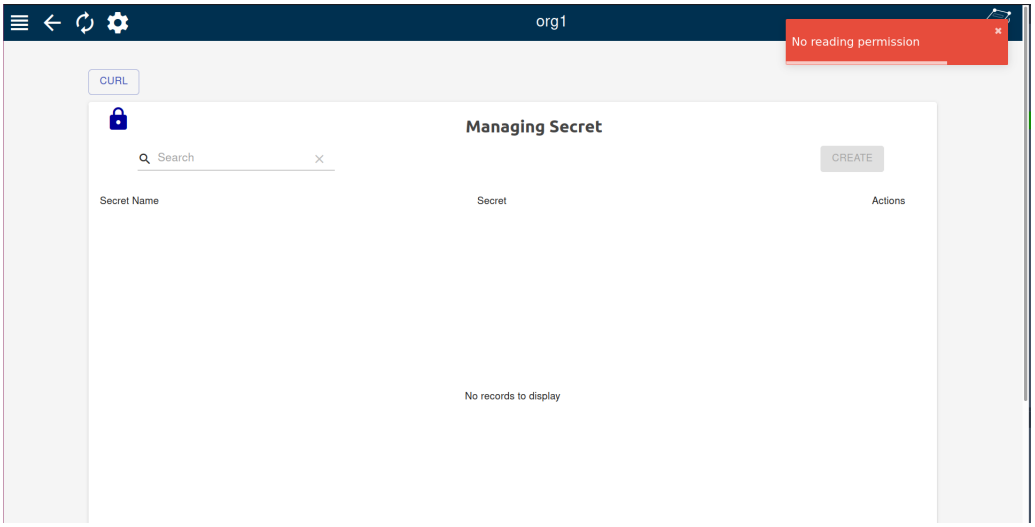
Fonte: Autoria própria.

Os mesmos passos podem ser efetuados para livros e bibliotecas, porém no caso dos livros eles podem possuir uma pessoa associada. Esta pessoa pode ser uma já cadastrada na rede assim como acabamos de reproduzir. Em uma biblioteca é possível relacionar os livros com a mesma, assim como pode-se definir categorias para controle interno da biblioteca.

O ativo nomeado SECRET é um ativo para mostrar a capacidade da rede de reter informações nas quais só podem ser exibidas e alteradas por certas organizações. Este ativo só pode ser criado e alterado pela Org2 e lido pela Org2 e Org3, a Org1 não tem acesso ao mesmo

portanto não recebe nenhuma informação ao clicar em sua listagem, o bloqueio de acesso ao ativo pode ser vizualizado na Figura 12.

Figura 12 – Tela de listagem do ativo Secret com bloqueio de acesso pela Organização 1



Fonte: Autoria própria.

5.4 DESENVOLVIMENTO

Para o desenvolvimento na rede *Blockchain* utilizando *Hyperledger Fabric*, utilizaremos o mesmo projeto que foi executado como exemplo na seção 5.3. Neste caso é necessário um pouco de conhecimento na linguagem de programação GO, já que todos os contratos inteligentes da rede são desenvolvidos por esse meio.

Além disso, é trabalhado com uma biblioteca para produção de código conhecida como *GoLedger CC-Tools*. Esta ferramenta auxilia na produção do código com maior facilidade já que traz algumas funções básicas pré-implementadas. Entre essas ferramentas também é incluso um pacote para implementação do código em produção utilizando a plataforma *GoFabric*, que permite trabalhar com a rede de forma privada e personalizada em ambientes de nuvem de forma simplificada.

Para navegar pelos arquivos do projeto e editar os códigos é utilizado o programa chamado *VSCode*, sua instalação é simples e detalhada no site *https://code.visualstudio.com/docs/setup/linux*. Para abrir o projeto no *VSCode* é só navegar com o terminal até a pasta *"cc-tools-demo"* e utilizar o comando:

```
code .
```

5.5 *ASSETS*

Assets, também conhecidos como ativos, são a representação das informações que serão utilizadas no *Hyperledger Fabric Channel* (*Blockchain ledger* ou somente *ledger*). Pode também ser comparado a uma tabela em um banco de dados relacional, já que são formados por propriedades com cada uma podendo ter um tipo específico de dados, incluindo também a possibilidade de fazer parte de um conjunto de chaves para os ativos.

Outra possibilidade de ativo é ser um *Hyperledger Fabric Private Data*, em que o seu conteúdo tem o registro de dados fora do ledger em um banco de dados transitório. São dados compartilhados entre um subconjunto de organizações, não contemplando toda a rede, e mesmo que uma organização fora deste conjunto receba os dados ela não possuiria a configuração para a sua leitura.

Alguns exemplos de ativos podem ser observados no capitulo anterior, que possuimos a Pessoa, Livro, Biblioteca e Segredo (como dado privado). Neste ponto utilizando o *VSCode* é possível editar os ativos já existentes que são encontrados em "chaincode/assettypes".

Para definirmos um ativo devemos além de criar um novo arquivo com "NomeDoAtivo.go"na pasta "assettypes", preenchê-lo com os seguintes campos:

- *Tag*: É um campo para utilização interna, definindo o nome para o ativo a ser referenciado pela REST API e pelo código. Neste campo não é permitido espaços ou caracteres especiais, é análogo ao nome de uma variável em C por exemplo;
- *Label*: Um texto livre que define o rótulo de como este ativo será nomeado em aplicativos externos, seu tipo de dado é *string*;
- *Description*: também com tipo de dado *string*, este campo é o que conterá a descrição do ativo para ser acessado por aplicativos externos;
- *Props*: Definido adiante em 5.5.1;
- *Readers*: Campo utilizado para definir dados privados de organizações, neste caso necessita de configuração adicional no arquivo "collections.json";

5.5.1 Propriedades dos *Assets*

Um conjunto de propriedades é necessário para que um ativo possa ser estruturado. Essas propriedades são um conjunto de informações do ativo que auxiliam no controle dos dados e qual o tipo de informação que cada ativo carrega. As propriedades de um ativo também são conhecidas por *Props* ou *Assets properties*. Assim listamos abaixo:

- *IsKey*: Campo booleano que indica se as propriedades do ativo é uma chave do mesmo;

- *Required*: Campo booleano que indica se as propriedades de um ativo requerem serem preenchidas obrigatoriamente;
- *ReadOnly*: Campo booleano que indica se as propriedades de um ativo são do modo apenas leitura após ser criado, ou seja, o campo de propriedades não pode ser alterado após criado;
- *DefaultValue*: Valor padrão a ser utilizado na propriedade do ativo, depende do tipo de dado do ativo para ser definido;
- *Writers*: Indica quais organizações podem criar ou alterar esse dado, caso seja uma propriedade chave este campo só poderá ser criado pela organização. Seu tipo de dado é uma lista de *strings*;
- *DataType*: Tipo de dado contido na propriedade. Os tipos padrões são *string, number, datetime* e booleano, porém tipos personalizados ainda podem ser definidos na pasta "chaincode/datatypes";
- *Validity*: Função para validação da propriedade, em que é sugerido utilizar funções simples de validação. Para serem utilizadas funções mais complexas é sugerido utilizar um tipo de dado personalizado

5.5.2 Exemplo de *Asset*

Alguns campos dos ativos podem não ficar muito claros apenas na teoria, por isso trago um exemplo utilizando o ativo *Person* para referência dos campos e principalmente das propriedades.

Listagem 1 – Exemplo de *Asset*

```
var Person = assets.AssetType{
    Tag:         "person",
    Label:       "Person",
    Description: "Personal data of someone",

    Props: []assets.AssetProp{
        {
            // Primary key
            Required: true,
            IsKey:    true,
            Tag:      "id",
            Label:    "CPF (Brazilian ID)",
            DataType: "cpf",                // Datatypes are
                                //identified at datatypes folder
            Writers:  []string{`org1MSP`}, // This means only org1
                              //can create the asset (others can edit)
        },
        {
            // Mandatory property
            Required: true,
            Tag:      "name",
            Label:    "Asset Name",
            DataType: "string",
            // Validate funcion
            Validate: func(name interface{}) error {
```

```
                nameStr := name.(string)
                if nameStr == "" {
                    return fmt.Errorf("name must be non-empty")
                }
                return nil
            },
        },
        {
            // Optional property
            Tag:      "dateOfBirth",
            Label:    "Date of Birth",
            DataType: "datetime",
        },
        {
            // Property with default value
            Tag:          "height",
            Label:        "Person's height",
            DefaultValue: 0,
            DataType:     "number",
        },
    },
}
```

Fonte: GoLedger cc-tools.

Como é visto no código, definir um ativo é simples. Além disso como suas propriedades são modulares qualquer alteração em caso de adição ou subtração de itens pode ser rapidamente efetuada no código base. No caso de inserção de novas propriedades não é necessário preocupar-se com os ativos já existentes caso tenha sido definido um valor padrão para este novo item, nos ativos que já existiam antes da inserção deste dado receberão o valor padrão por definição.

5.6 *TRANSACTIONS*

As *Transactions* (transações) definidas pela biblioteca *GoLedger CC-Tools* são métodos implementados em *GoLang* para modificar os ativos utilizando de uma *Blockchain ledger (Hyperledger Fabric Channel)*. Para um ativo ser modificado é necessário uma operação via *chaincode*, ou seja, uma operação na rede *Hyperledger* que contenha a confirmação das organizações inclusas (**endorsing peers**).

Utilizando a *GoLedger CC-Tools* já possuimos algumas transações pré definidas assim como *CreateAsset, UpdateAsset, DeleteAsset, ReadAsset, ReadAssetHistory* e *Search* que permitem respectivamente criar ativos, atualizá-los, deletá-los, ler seu conteúdo, ler seu histórico de alterações e buscar a lista de ativos. Nesta biblioteca também é permitido criar transações personalizadas, além disso, no exemplo que estamos utilizando como base já possui transações personalizadas que podem ser encontradas em "chaincode/txdefs".

Assim como nos ativos, as transações possuem campos a serem preenchidos. Dentro do arquivo "txdefs"precisamos definir alguns elementos para a nossa transação ser válida.

- *Tag*: Define o nome da transação que será utilizado internamente no código e pelos *endpoits* da *REST API*. Suas propriedades de definição são semelhantes a *Tag* dos ativos, porém caso possuam o mesmo nome de uma transação predefinida substituirá a função original. Seu campo de dados também é uma *string*;

- *Label*: Campo *string* de texto livre. Utilizado para definir o rótulo que será utilizado por aplicações externas;
- *Description*: Também um campo *string* de texto livre, é utilizado por aplicativos externos para mostrar a descrição da transação;
- *Method*: Pode ser dos tipos "POST", "GET", "PUT"e "DELETE". É utilizado pela REST API para identificar o método de transação de dados;
- *Callers*: É utilizado para definir quais organizações podem utilizar desta transação;
- *Args*: Definido adiante em subseção 5.6.1
- *Routine*: Código fonte da transação, utilizado para fazer o processamento dos dados na rede;

5.6.1 Argumentos da *Transaction*

Os argumentos de uma transação podem ser considerados como um conjunto personalizado de entrada, os campos para se definir um argumento são os seguintes:

- *Tag*: Campo do tipo *string* que não permite caracteres especiais nem espaços. Utilizado no código para ser referenciado e definir o nome do argumento;
- *Label*: Texto livre do tipo *string*. Utilizado de rótulo para aplicativos externos;
- Description: Texto livre do tipo *string*. Utilizado por aplicativos externos para descrever o argumento;
- *Required*: Campo booleano. Utilizado para definir se o argumento é obrigatório ou não;
- *DataType*: Tipo de dado do argumento. Os tipos padrões são *string, number, datetime* e booleano, porém ainda podem ser utilizados tipos personalizados assim como nos ativos;

5.6.2 Exemplo de *Transaction*

Será representado também um exemplo de transação utilizando uma das transações personalizadas inclusa neste exemplo. De forma que o entendimento da teoria apresentada anteriormente fique facilitado, abaixo é utilizado a transação "createNewLibrary"para elucidar visualmente a montagem da função.

Listagem 2 – Exemplo de *Transaction*

```
var CreateNewLibrary = tx.Transaction{
    Tag:         "createNewLibrary",
    Label:       "Create New Library",
    Description: "Create a New Library",
    Method:      "POST",
    Callers:     []string{"$org3MSP"},
                    // Only org3 can call this transaction

    Args: []tx.Argument{
        {
            Tag:         "name",
            Label:       "Name",
            Description: "Name of the library",
            DataType:    "string",
            Required:    true,
        },
    },
    Routine: func(stub shim.ChaincodeStubInterface,
                    req map[string]interface{}) ([]byte, errors.ICCError) {
        name, ok := req["name"].(string)
        if !ok {
            return nil, errors.WrapError(nil,
                                "Parameter name must be string")
        }

        libraryMap := make(map[string]interface{})
        libraryMap["@assetType"] = "library"
        libraryMap["name"] = name

        libraryAsset, err := assets.NewAsset(libraryMap)
        if err != nil {
            return nil, errors.WrapError(err,
                                    "Failed to create a new asset")
        }

        // Save the new library on channel
        _, err = libraryAsset.PutNew(stub)
        if err != nil {
            return nil, errors.WrapError(err,
                                    "Error saving asset on blockchain")
        }

        // Marshal asset back to JSON format
        libraryJSON, nerr := json.Marshal(libraryAsset)
        if nerr != nil {
            return nil, errors.WrapError(nil,
                                "failed to encode asset to JSON format")
        }

        return libraryJSON, nil
    },
}
```

Fonte: GoLedger cc-tools.

A partir deste ponto já se é possível iniciar o desenvolvimento próprio de elementos e funções para uma rede *Blockchain Hyperledger Fabric*, utilizando da biblioteca *GoLedger CC-Tools*.

5.7 DESENVOLVENDO O *CHAINCODE*

Utilizando da base de testes fornecida pela *CC-Tools-Demo*, será criado ativos e transações próprias com base em uma aplicação de foco em cartório, na qual teremos ativos como pessoas e propriedades, transações como transferência de posse e fundos monetários e também operações que só possam ser executadas por organizações específicas. Esta aplicação será voltada totalmente para o desenvolvimento do código na parte de *Blockchain*, portanto não será

implementada nenhuma iteração gráfica relacionada ao usuário final.

5.7.1 Desenvolvendo *Assets*

Utilizando o terminal diretamente na pasta "cc-tools-demo", é executado o comando para abrir o *VSCode*. Já no aplicativo na localização "chaincode/assettypes"é iniciado um novo arquivo "propriedade.go". Este é um ativo que simboliza os itens que um indivíduo pode possuir, qualquer item com quesito de posse que possa ser documentado, e diferenciado de outros do mesmo tipo, tem o direito de ser este item.

Primeiro é definido uma *Tag* para o produto, por convenção as *tags* e *labels* serão definidas pela mesma palavra, porém as *tags* iniciarão com letra minúscula e as *labels* com letra maiúscula. Portanto a *Tag* e *Label* para este ativo serão definidas como "produto"e "Produto"respectivamente. A *Description* é o campo que aplicações externas utilização como descrição do que será inserido, como nosso produto é genérico para qualquer item devemos utilizar uma descrição abrangente como "Item rastreável de código único que utiliza identificador de posse".

Para definir as propriedades desse ativo deve-se buscar primeiramente qual será o nível de detalhamento do produto. Caso queira definir de forma mais detalhada este passo pode ser o mais demorado, porém em nosso caso como a aplicação não tem necessidade de ser repetitiva serão definidas algumas poucas propriedades apenas para identificação de uma possível posse.

A primeira propriedade a ser definida será a chave primária, que se dará por um identificador exclusivo do item, está propriedade deve conter os elementos *Required* e *Iskey* como *true*. Sua *Tag* e *Label* serão respectivamente "identificador"e "Identificador". Seu tipo de dado é *string*, pois pode conter tanto números, quanto letras e caracteres especiais. Não será criado o campo *Writers* neste momento, pois as organizações ainda não foram definidas. Já o campo *Validate* deverá conter uma validação para que o valor inserido não seja vazio.

A chave primaria deste ativo será a propriedade mais complexa já que possui caráter de identificação e controle, outras duas propriedades serão adicionadas, são as propriedades Descrição e Proprietário. Ambas serão do tipo de dado *string* e não serão chaves. A propriedade descrição será Requerida obrigatoriamente, já a propriedade proprietário só será preenchida quando este ativo for pertence de alguma pessoa.

Portanto nosso primeiro ativo ficou neste formato:

Listagem 3 – Ativo desenvolvido

```
1  var Produto = assets.AssetType{
2      Tag:          "produto",
3      Label:        "Produto",
4      Description: "Item rastreável de código único
5                   que utiliza identificador de posse",
6
7      Props: []assets.AssetProp{
8          {
9              // Chave primária
10             Required: true,
```

```
            IsKey:    true,
            Tag:      "identificador",
            Label:    "Identificador",
            DataType: "string",
            Validate: func(identificador interface{}) error {
                identificadorStr := identificador.(string)
                if identificadorStr == "" {
                    return fmt.Errorf("identificador não
                                        pode ser vazio")
                }
                return nil
            },
        },
        {
            // Descrição do produto
            Required: true,
            Tag:      "descricao",
            Label:    "Descrição",
            DataType: "string",
            Validate: func(name interface{}) error {
                nameStr := name.(string)
                if nameStr == "" {
                    return fmt.Errorf("Descrição não
                                        pode ser vazio")
                }
                return nil
            },
        },
        {
            // propriedade opcional, dono do produto
            Tag:      "proprietario",
            Label:    "Proprietário",
            DataType: "string",
        },
    },
}
```

Fonte: Autoria própria

O próximo ativo a ser desenvolvido é "pessoa.go". Este ativo será definido com *Tag* e *Label* "pessoa"e "Pessoa"respectivamente. *Description* será definido como "Pessoa, propriedades e financeiro". Já as propriedades serão 4, A chave principal, numérica, requisitada, não nula e apenas leitura identificada com o CPF da pessoa que foi cadastrada. A segunda propriedade com o tipo de dado sendo uma lista de Produtos, não será requisitado, não será somente leitura, poderá ser vazia que possui *Tag* e *Label* como "posses"e "Posses"respectivamente. A terceira propriedade será requerida, com valor padrão como zero, *Tag* e *Label* "dinheiro"e "Dinheiro"respectivamente, seu tipo de dado será numérico. A última das propriedades terá a *Tag* e *Label* respectivamente como "aparencia"e "Aparência", podendo ser vazia. As *Descriptions* serão relacionadas as propriedades trazendo uma breve descrição de sua utilidade.

Após produzidos os Ativos deve-se navegar com o terminal até a pasta chaincode e utilizar um comando para se verificar a sintaxe dos arquivos ".go". O comando a ser utilizado é:

```
go vet
```

Caso tudo ocorra bem, deve-se inserir os novos ativos no arquivo "assetTypeList.go". Utilizando como base o arquivo que já existia no exemplo anterior o arquivo deve ficar neste formato:

Listagem 4 – Arquivo de gerenciamento de ativos

```
package main

import (
    "github.com/goledgerdev/cc-tools-demo/chaincode/assettypes"
    "github.com/goledgerdev/cc-tools/assets"
)

var assetTypeList = []assets.AssetType{
    assettypes.Person,
    assettypes.Book,
    assettypes.Library,
    assettypes.Secret,
    assettypes.Produto,
    assettypes.Pessoa,
}
```

Fonte: Autoria própria

Caso ocorra algum erro de sintaxe deve-se verificar os ativos criados para que fiquem no padrão pedido pelo GoLang. Após o procedimento ser efetuado executamos novamente o comando para verificar a sintaxe. Quando são feitas alterações no *chaincode* existe mais um procedimento a ser efetuado, porém ele só será executado após nossas transações estarem completas.

5.7.2 Desenvolvendo *Transactions*

Após os Ativos serem criados e possuirem sua sintaxe verificada, deve-se dar início as transações que manipularão estes Ativos. No caso da nossa aplicação serão desenvolvidas transações que criam e listam nossos ativos, além de uma transação para exibir o número de posses de um ativo do tipo pessoa. Para criarmos nossas transações deve-se navegar até a pasta "chaincode/txdefs"pelo *VSCode*.

A primeira transação a ser criada será chamada de "atualizarPropriedade.go", será a função para alterar a pessoa associada ao produto. está transação será do método "POST", pois envia um dado para a rede, este dado pode vir tanto de uma interface visual quanto de uma resposta de API.

Nas transações utilizarei o mesmo processo que nos ativos referente aos campos *Tag* e *Label*, de modo que seja simplificado o conteúdo menos revelante a ser descrito. Portanto sua *Tag* e *Label* ficariam respectivamente "atualizarPropriedade"e "Atualizar Propriedade". A *Description* desta transação ficará como "Atualiza proprietário da propriedade", e como mencionado anteriormente o *Method* ficará como "POST". O campo *Callers* é definido para que todas as organizações possam chamar a transação no momento, seu conteúdo possui a seguinte expressão:

```
"[]string{`$org\dMSP`}"
```

Já para definir os argumentos necessários, é utilizado os elementos que são alterados. Neste caso será utilizado a lista de propriedades, para saber qual propriedade será alterada e também a lista de pessoas para poder selecionar a quem será associado aquele item. Então para

finalizar foi definido o campo *Routine*, foi implementada uma função que recebe os itens da tela em formato *JSON*, separa as variáveis que consistem na propriedade a ser alterada e na pessoa que será definida como proprietário, verifica se não são nulos e prossegue com o código. Após a verificação é feita a associação da pessoa ao item, é feita a atualização e então o empacotamento do ativo de volta para *JSON*.

Após serem implementadas as atividades descritas o código da transação ficou neste formato:

Listagem 5 – Transação desenvolvida

```
package txdefs

import (
    "encoding/json"

    "github.com/goledgerdev/cc-tools/assets"
    "github.com/goledgerdev/cc-tools/errors"
    sw "github.com/goledgerdev/cc-tools/stubwrapper"
    tx "github.com/goledgerdev/cc-tools/transactions"
)

// Atualiza o proprietário da propriedade
// POST Method
var AtualizarPropriedade = tx.Transaction{
    Tag:         "atualizarPropriedade",
    Label:       "Atualizar Propriedade",
    Description: "Atualiza proprietário da propriedade",
    Method:      "POST",
    Callers:     []string{`$org\dMSP`}, // todas as organizações podem
    //                                     chamar esta transação

    Args: []tx.Argument{
        {
            Tag:         "propriedade",
            Label:       "Propriedade",
            Description: "Propriedade",
            DataType:    "->produto",
            Required:    true,
        },
        {
            Tag:         "proprietario",
            Label:       "Proprietário",
            Description: "Novo proprietário do item",
            DataType:    "->pessoa",
        },
    },
    Routine: func(stub *sw.StubWrapper, req map[string]interface{})
    ([]byte, errors.ICCError) {
        propriedadeChave, ok := req["propriedade"].(assets.Key)
        if !ok {
            return nil, errors.WrapError(nil, "Parâmetro
                                              deve ser um Ativo")
        }
        proprietarioChave, ok := req["proprietario"].(assets.Key)
        if !ok {
            return nil, errors.WrapError(nil, "Parâmetro
                                    proprietário deve ser um Ativo")
        }

        // Retorna o livro do canal
        propriedadeAtivo, err := propriedadeChave.Get(stub)
        if err != nil {
            return nil, errors.WrapError(err, "Falha ao buscar o Ativo
                                                        no ledger")
        }
        propriedadeMap := (map[string]interface{})(*propriedadeAtivo)

        // Retorna a pessoa do canal
        proprietarioAtivo, err := proprietarioChave.Get(stub)
        if err != nil {
            return nil, errors.WrapError(err, "Falha ao buscar o Ativo
                                                        no ledger")
```

```
63          }
64          proprietarioMap := (map[string]interface{})(*proprietarioAtivo)
65
66          chaveProprietarioAtualizado := make(map[string]interface{})
67          chaveProprietarioAtualizado["@assetType"] = "pessoa"
68          chaveProprietarioAtualizado["@key"] = proprietarioMap["@key"]
69
70          // Atualiza os dados
71          propriedadeMap["proprietario"] = chaveProprietarioAtualizado
72
73          propriedadeMap, err = propriedadeAtivo.Update(stub, propriedadeMap)
74          if err != nil {
75             return nil, errors.WrapError(err, "Falha ao atualizxar o Ativo")
76          }
77
78          // Empacota o Ativo novamente para o formato JSON
79          propriedadeJSON, nerr := json.Marshal(propriedadeMap)
80          if nerr != nil {
81              return nil, errors.WrapError(err, "Falha ao empacotar o Ativo")
82          }
83
84          return propriedadeJSON, nil
85      },
86  }
```

Fonte: Autoria própria

Com o código da transação concluído é possível complementar o arquivo "tx-List.go"adicionando a entrada da nova transação personalizada. Este arquivo é encontrando dentro da pasta "chaincode"e após adicionado o novo campo seu conteúdo fica desta forma:

Listagem 6 – Arquivo de gerenciamento de transações

```
1   package main
2
3   import (
4           txdefs "github.com/goledgerdev/cc-tools-demo/chaincode/txdefs"
5
6           tx "github.com/goledgerdev/cc-tools/transactions"
7   )
8
9   var txList = []tx.Transaction{
10          tx.CreateAsset,
11          tx.UpdateAsset,
12          tx.DeleteAsset,
13          txdefs.CreateNewLibrary,
14          txdefs.GetNumberOfBooksFromLibrary,
15          txdefs.UpdateBookTenant,
16          txdefs.AtualizarPropriedade,
17  }
```

Fonte: Autoria própria

Novamente para testar a sintaxe do código utilizo no terminal o comando:

```
go vet
```

Como não foram definidas organizações específicas para o código criado, é possível atualizar o *chaincode* para as organizações e realizar um teste da função criada. Está biblioteca (*CC-Tools*) permite ter a visualização em tela dos elementos criados de forma que não seja necessário nenhuma implementação de código visual, portanto é elevado a praticidade no caso de uso comum.

5.7.3 Atualizando o *chaincode*

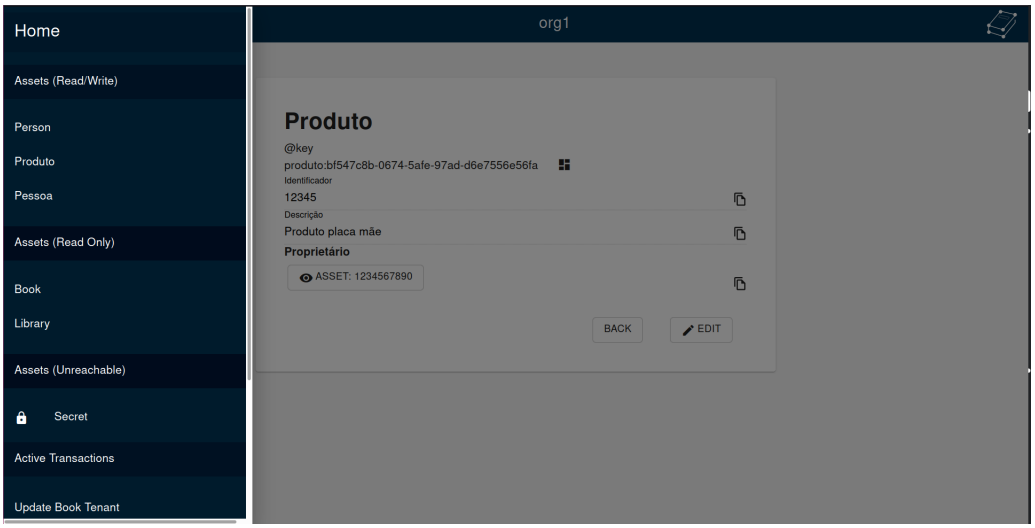
Após alterado o *chaincode* precisa ser atualizado nos *peers*. Neste caso também é disponibilizado um *script* que faz esta alteração em todas as organizações. Como é necessário recompilar todo o *chaincode* está ação pode ser mais lenta em casos de códigos muito complexos. O *script* a ser utilizado depende de um parâmetro, a versão, este campo nunca deve se repetir pois indica que o código foi atualizado a uma versão superior. Abaixo deixo o comando a ser utilizado no terminal, visando a atualização do *chaincode*:

```
sysadmin@localhost:~$ ./upgradeCC.sh 0.2

Upgrade chaincode to version 0.2
{"status":"SUCCESS"}
sysadmin@localhost:~$
```

Recebendo a mensagem de sucesso é possível acessar novamente as Organizações e verificar que as alterações já estão disponíveis para serem testadas. Ao acessar a Org1, é possível ser verificado que no menu lateral já está disponível os novos Ativos e a nova transação assim como é mostrado na Figura 13.

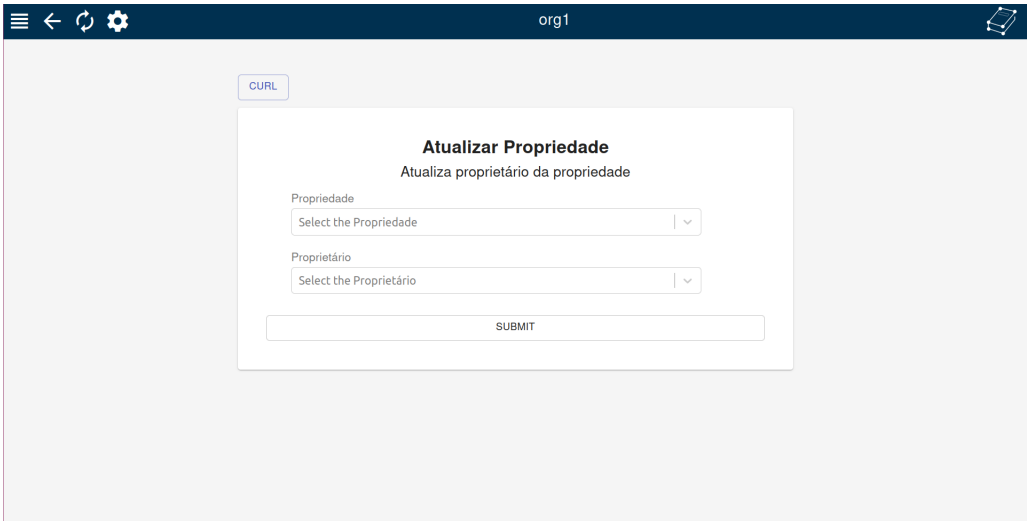
Figura 13 – Tela exibindo o Menu lateral atualizado



Fonte: Autoria própria.

Neste momento é possível efetuar um teste inserindo um novo dado para os ativos Pessoa e Produto, além de também ser possível Utilizar a transação Atualizar Propriedade para associar a Pessoa ao Produto, como pode ser visto na Figura 14.

Figura 14 – Tela da transação Atualizar Propriedade



Fonte: Autoria própria.

5.8 DEFININDO AS ORGANIZAÇÕES

Ao definir uma organização para um ativo ou transação também define quem pode acessar, alterar, criar e excluir os elementos. Um exemplo que já existe nesta rede de testes é o ativo *Secret*, que só pode ser criado e excluído pela Org2 e pode ser lido e editado pela Org2 e Org3. A Org1 não tem acesso a este ativo e em sua interface só recebe um *hash* que informa a não disponibilidade da informação.

Após realizado os testes dos nossos ativos e transações é possível definir quem pode acessar e editar esses ativos. No caso dos ativos esta configuração fica em suas *Props* num campo chamado *Writers*, para cada propriedade do ativo é possível definir uma ou mais organizações que podem editá-lo. Caso este campo não exista ou esteja vazio qualquer organização poderá alterar este elemento.

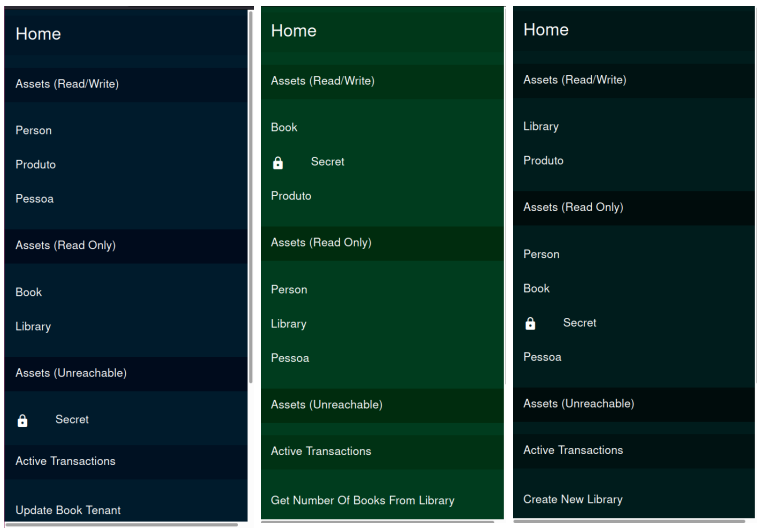
Nesta aplicação é definido que apenas a Org1 pode editar o ativo Pessoa, por enquanto a Org2 e Org3 podem ver o elemento mas não é possível a partir das mesmas alterar as informações desse elemento. Para isso é incluso o campo *Writers* em todas as *Props* do ativo Pessoa, e seu valor é definido como:

```
[]string{`org1MSP`},
```

A partir da edição de todas as propriedades do ativo Pessoa, é possível executar a verificação de sintaxe do código e então sua atualização. Lembrando sempre que a atualização deve ser executada sempre com uma versão diferente à anterior. Após atualizado é possível verificar nas páginas das organizações que somente na Org1 o ativo Pessoa está na subcategoria *Assets (Read/Write)*, como é exibido na Figura 15.

Figura 15 – Menu lateral das Organizações exibindo o ativo Pessoa em seções diferentes

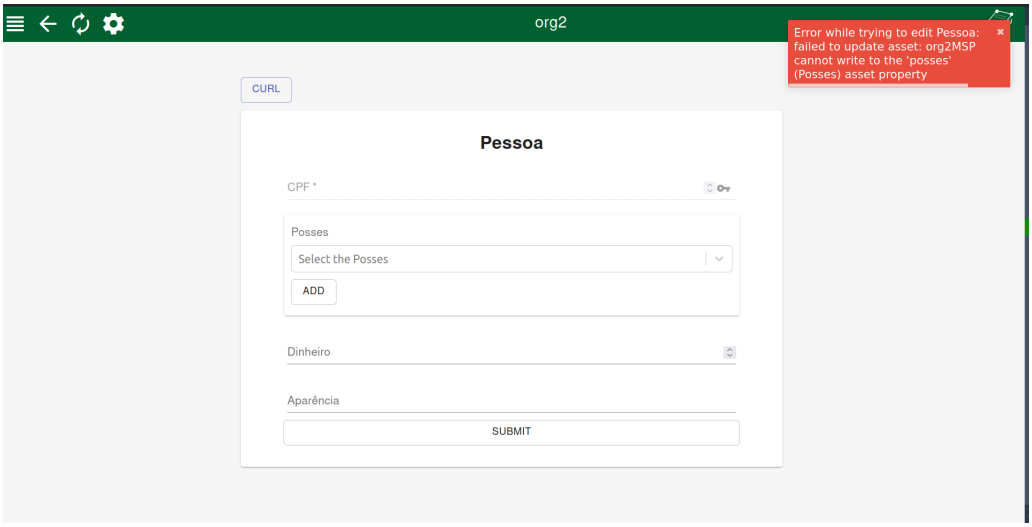
(a) Org1 **(b) Org2** **(c) Org3**



Fonte: Autoria própria.

É possível editar o ativo Pessoa com a Org1, além da possibilidade de criar ativos do tipo Pessoa a partir desta organização. Porém nas Org2 e Org3 este ativo está somente como leitura, e o botão para criar novo ativo fica desabilitado, no entanto é possível clicar no botão de edição de um ativo existente. Ao clicar em edição do ativo é exibida a tela de edição, porém caso alguma alteração seja feita e depois clicado em *SUBMIT* a rede retorna um erro como é exibido na Figura 16.

Figura 16 – Tela de erro ao tentar alterar ativo sem permissão



Fonte: Autoria própria.

Também é possível implementar ativos visíveis em apenas algumas organizações, como é o caso do ativo *Secret*. Para executar esse procedimento é necessário adicionar o campo *Readers* ao ativo, este campo contém uma lista de *strings* com as organização que podem visualiza-

lo. Neste caso o acesso será permitido somente a Org1 e Org2, portanto o campo *Readers* deve ser preenchido com:

```
[]string{"org1MSP", "org2MSP"},
```

Além deste passo o arquivo "collections.json"deve ser editado para informar a rede que este ativo possui o campo *Readers*. Sua configuração ficará semelhante a que já existe para o ativo *Secret*, apenas alterando a informação do campo *name* pela *Tag* do ativo Pessoa e também alterando as organizações pela Org1 e Org2. Após alterado o arquivo fica como mostrado na Listagem 7

Listagem 7 – Arquivo "collections.json"contendo o código auxiliar para o ativo Pessoa

```
[
  {
    "name": "secret",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive": 1000000,
    "memberOnlyRead": true,
    "policy": {
      "identities": [
        {
          "role": {
            "name": "member",
            "mspId": "org2MSP"
          }
        },
        {
          "role": {
            "name": "member",
            "mspId": "org3MSP"
          }
        }
      ],
      "policy": {
        "1-of": [
          {
            "signed-by": 0
          },
          {
            "signed-by": 1
          }
        ]
      }
    }
  },
  {
    "name": "pessoa",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive": 1000000,
    "memberOnlyRead": true,
    "policy": {
      "identities": [
        {
          "role": {
            "name": "member",
            "mspId": "org1MSP"
          }
        },
        {
          "role": {
            "name": "member",
            "mspId": "org2MSP"
          }
        }
      ],
      "policy": {
        "1-of": [
          {
```

```
                "signed-by": 0
            },
            {
                "signed-by": 1
            }
        ]
    }
  }
]
```

Fonte: Autoria própria

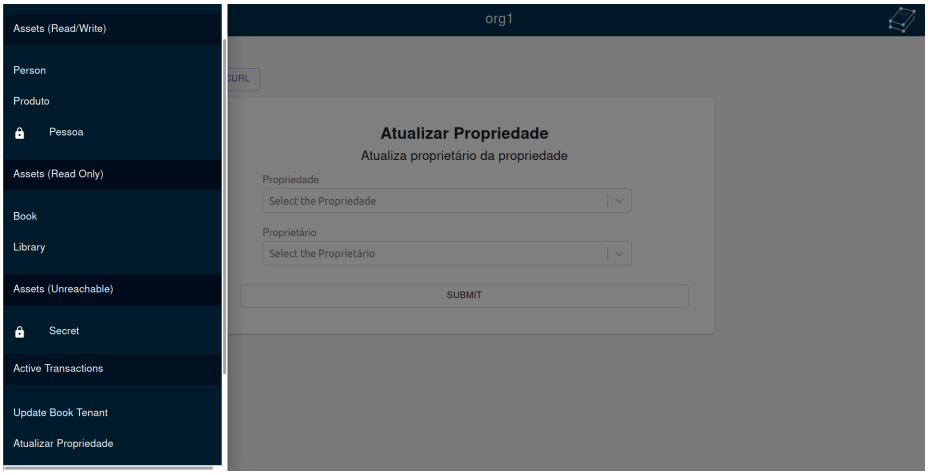
Após esses passos já é possível recompilar o *chaincode* para exibir o ativo Pessoa somente para Org1 e Org2. Porém também é possível definir permissões de acesso a transações, portanto previamente a recompilação e testes foi efetuado a alteração de permissão para a transação "atualizarPropriedade". Para tal é necessário alterar os dados do elemento *Callers*, em que foi definido anteriormente na Listagem 2, seu valor deve ser sobrescrito por:

```
[]string{"$org1MSP"},
```

A Org2 será um elemento de apenas consulta para o ativo Pessoa, não podendo efetuar edições a partir desta organização, em um exemplo na realidade é como se a organização 1 fosse um cartório, podendo cadastrar e alterar dados e posses de pessoas e a organização 2 fosse um elemento de consulta no qual não é possível fazer alterações porém é possível visualizar os dados cadastrados. Neste caso a organização 3 é um elemento que não pode nem alterar os dados e nem visualizá-los, seria como um elemento externo ao cartório que não possui estas permissões.

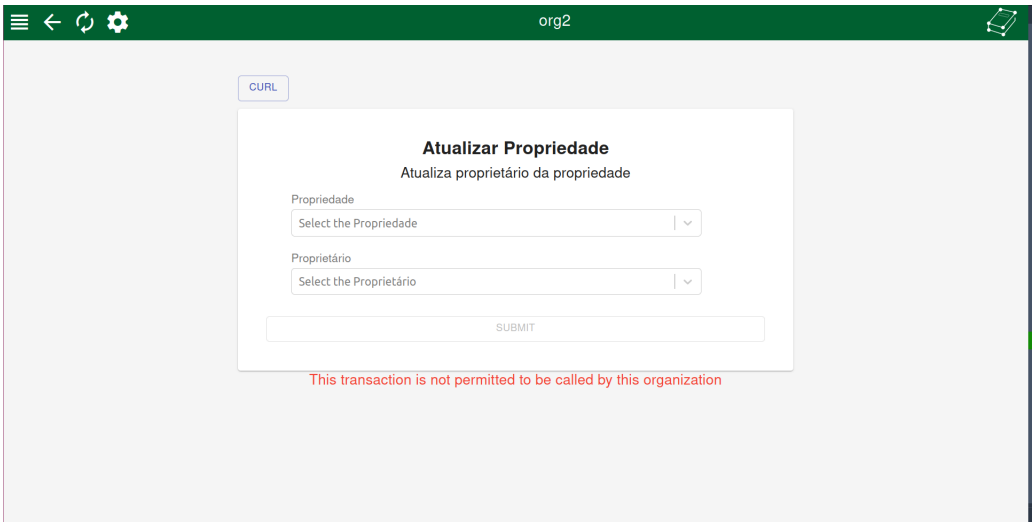
É possível então realizar uma nova verificação de sintaxe, e compilação do *chaincode*, para executar os testes de acessos das organizações. Obtendo o status de sucesso ao atualizar o *chaincode* é possível verificar nas páginas das organizações que, Org1 tem acesso a escrita do ativo Pessoa além de ter acesso a transação "alterarPropriedade". Org2 e Org3 não tem acesso a transação criada, Org2 tem somente acesso de leitura para o ativo Pessoa e Org3 não possui acesso ao mesmo. Estes exemplos podem ser visualizados nas Figura 17, Figura 18 e Figura 19.

Figura 17 – Tela da organização 1 com total acesso ao ativo e transação



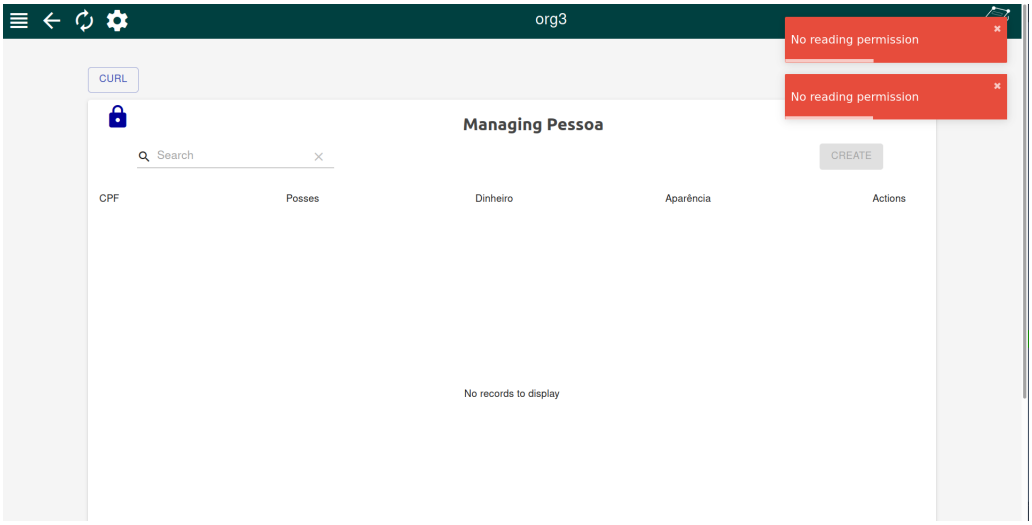
Fonte: Autoria própria.

Figura 18 – Tela da organização 2 sem acesso a transação



Fonte: Autoria própria.

Figura 19 – Tela da organização 3 sem acesso ao ativo Pessoa



Fonte: Autoria própria.

6 CONCLUSÃO

Inicialmente foi proposto o desenvolvimento de uma aplicação utilizando um *framework Blockchain*, além de um comparativo dentre os mesmos e um guia de desenvolvimento. Para tal também foi utilizado como material de referência informações sobre *Blockchain* e seus *frameworks*. Todas as ferramentas informadas foram utilizadas em algum momento no desenvolvimento deste trabalho, com um destaque principal aos materiais de pesquisa.

Algumas vantagens das tecnologias utilizadas são encontradas na área de desenvolvimento, na qual a biblioteca *CC-Tools* possibilitou um desenvolvimento mais rápido e facilitado da aplicação. No entanto durante o comparativo entre os *frameworks* a compilação de dados se tornou bem ampla, pois cada *framework* analisado possui suas particularidades. Para solucionar este problema foram buscados termos que abrangem uma gama maior de opções, podendo então realizar o comparativo dentre os *frameworks*.

Este trabalho pode contribuir com a comunidade *Blockchain* trazendo um tutorial simplificado para o desenvolvimento de uma aplicação, além de contar com uma tabela comparativa entre alguns dos *frameworks* mais famosos. Esta tabela pode ser utilizada para definição de quais *framework* são mais indicados a sua aplicação, tendo a possibilidade de focar o desenvolvimento em somente uma área.

Como foi proposto no início deste trabalho foi feita um comparativo dos frameworks da rede *Blockchain*, além do desenvolvimento de uma aplicação e um guia para desenvolvimento. Para cumprir o objetivo específico do comparativo foi elaborado uma tabela comparando algumas características encontradas em todos os *frameworks*, além de duas tabelas auxiliares informando pontos vantajosos e desvantajosos de cada *framework*.

A análise e desenvolvimento da aplicação foram efetuados após o primeiro objetivo ser concluído. Pois utilizando da tabela o caso de uso proposto pode ser facilmente associado a um *framework*, neste caso o *Hyperledger Fabric*. Em sequência, o desenvolvimento da aplicação foi iniciado e utilizando de uma biblioteca chamada *CC-Tools* em que foram criados os códigos, organizações, transações e ativos; os quais cumpriram o caso de uso proposto para a aplicação.

Em conjunto ao desenvolvimento da aplicação foi criado o guia para o desenvolvimento. Tanto o guia quanto o desenvolvimento estão integrados em uma única seção, pois utilizando deste método é eliminado duplicidade de informações ajudando também a compreender melhor o passo a passo executado. Acredita-se que todos os objetivos propostos tenham sido alcançados satisfatoriamente do que havia sido proposto.

REFERÊNCIAS

AGBO, Cornelius C; MAHMOUD, Qusay H. Comparison of blockchain frameworks for healthcare applications. **Internet Technology Letters**, Wiley Online Library, v. 2, n. 5, p. e122, 2019. Citado 2 vezes nas páginas 27 e 28.

ANTONOPOULOS, Andreas M; WOOD, Gavin. **Mastering ethereum: building smart contracts and dapps**. [S.l.]: O'reilly Media, 2018. Citado 2 vezes nas páginas 19 e 20.

BASET, S.A. *et al.* **Hands-On Blockchain with Hyperledger: Building decentralized applications with Hyperledger Fabric and Composer**. [S.l.]: Packt Publishing, 2018. ISBN 9781788996044. Citado na página 19.

BENJI, Mariya; SINDHU, M. A study on the corda and ripple blockchain platforms. In: **Advances in big data and cloud computing**. [S.l.]: Springer, 2019. p. 179–187. Citado na página 28.

BROWN, Richard Gendal. The corda platform: An introduction. **Retrieved**, v. 27, p. 2018, 2018. Citado na página 22.

BROWN, Richard Gendal *et al.* Corda: an introduction. **R3 CEV, August**, v. 1, p. 15, 2016. Citado na página 22.

CARDANO, ADA. Capıtulo 7. **Tecnologias de Informaçao de Suporteas Criptomoedas**, p. 53, 2021. Citado na página 28.

CLINCY, Victor; SHAHRIAR, Hossain. Blockchain development platform comparison. In: IEEE. **2019 IEEE 43rd annual computer software and applications conference (COMPSAC)**. [S.l.], 2019. v. 1, p. 922–923. Citado na página 28.

DATTANI, Janvi; SHETH, Harsh. Overview of blockchain technology. **Asian Journal of Convergence in Technology**, v. 5, n. 1, p. 1–3, 2019. Citado 2 vezes nas páginas 16 e 28.

GARRIGA, Martin *et al.* Blockchain and cryptocurrencies: A classification and comparison of architecture drivers. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 33, n. 8, p. e5992, 2021. Citado na página 28.

HYPERLEDGER. **Hyperledger – Open Source Blockchain Technologies**. 2020. Disponível em: https://www.hyperledger.org/. Citado 2 vezes nas páginas 15 e 21.

KAMMERER, Tommy. **Cardano – Making The World Work Better For All**. 2022. Disponível em: https://cardano.org/. Citado na página 23.

KOLVART, Merit; POOLA, Margus; RULL, Addi. Smart contracts. In: **The Future of Law and etechnologies**. [S.l.]: Springer, 2016. p. 133–147. Citado na página 18.

NADIR, Rana M. Comparative study of permissioned blockchain solutions for enterprises. In: IEEE. **2019 International Conference on Innovative Computing (ICIC)**. [S.l.], 2019. p. 1–6. Citado na página 28.

NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system. **Decentralized Business Review**, p. 21260, 2008. Citado na página 28.

PELT, RL van. **Blockchain governance: A framework for analysis and comparison**. 2019. Dissertação (Mestrado) — University Utrecht, 2019. Citado na página 28.

QUASIM, Mohammad Tabrez *et al.* Blockchain frameworks. In: **Decentralised Internet of Things**. [S.l.]: Springer, 2020. p. 75–89. Citado 2 vezes nas páginas 18 e 28.

SAJANA, P; SINDHU, M; SETHUMADHAVAN, M. On blockchain applications: hyperledger fabric and ethereum. **International Journal of Pure and Applied Mathematics**, v. 118, n. 18, p. 2965–2970, 2018. Citado 2 vezes nas páginas 16 e 28.

SCHWARTZ, David *et al.* The ripple protocol consensus algorithm. **Ripple Labs Inc White Paper**, v. 5, n. 8, p. 151, 2014. Citado na página 23.

____. **Ripple - Business impact, powered by crypto**. 2022. Disponível em: https://www.ripple.com/. Citado 2 vezes nas páginas 23 e 24.

SIIM, Janno. Proof-of-stake. In: **Research Seminar in Cryptography**. [S.l.: s.n.], 2017. Citado na página 18.

STEVENS, Marc *et al.* **On collisions for MD5**. 2007. Citado na página 15.

TAVARES, Bruno; CORREIA, F Figueiredo; RESTIVO, André. A survey on blockchain technologies and research. **Journal of Information Assurance and Security**, v. 14, p. 118–128, 2019. Citado na página 16.

ULRICH, Fernando. **Bitcoin: a moeda na era digital**. [S.l.]: LVM Editora, 2014. Citado na página 20.

VALENTA, Martin; SANDNER, Philipp. Comparison of ethereum, hyperledger fabric and corda. **ebook] Frankfurt School, Blockchain Center**, 2017. Citado 2 vezes nas páginas 27 e 28.

WOOD, Gavin *et al.* Ethereum: A secure decentralised generalised transaction ledger. **Ethereum project yellow paper**, v. 151, n. 2014, p. 1–32, 2014. Citado 2 vezes nas páginas 20 e 28.

YAGA, Dylan *et al.* Blockchain technology overview. **arXiv preprint arXiv:1906.11078**, 2019. Citado na página 15.

YAMASHITA, Kazuhiro *et al.* Potential risks of hyperledger fabric smart contracts. In: **2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)**. [S.l.: s.n.], 2019. p. 1–10. Citado na página 31.

ZAND, Matt; WU, Xun Brian; MORRIS, Mark Anthony. **Hands-On Smart Contract Development with Hyperledger Fabric V2**. [S.l.]: "O'Reilly Media, Inc.", 2021. Citado na página 31.