Universidade Federal de São Carlos – UFSCar
Centro de Ciências Exatas e de Tecnologia – CCET
Departamento de Computação – DC
Trabalho de Conclusão de Curso – TCC

Fernando Salhab Doria Ricardo

Rede Blockchain para rastreabilidade de queijos artesanais no Brasil utilizando Hyperledger Fabric

São Carlos - SP
2024

Fernando Salhab Doria Ricardo

Rede Blockchain para rastreabilidade de queijos artesanais no Brasil utilizando Hyperledger Fabric

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação da Universidade Federal de São Carlos, como parte dos requisitos para a conclusão da graduação em Engenharia de Computação.

Orientador: Prof. Dr. Fredy J. Valente

São Carlos - SP
2024

Ao meu querido tio Alemar (in memoriam),
exemplo de ser humano íntegro e ético.

Agradecimentos

Agradeço profundamente aos meus pais, Rogério e Elisete, por todo o apoio, amor e confiança incondicional ao longo da minha vida, que foram essenciais para que eu chegasse até aqui. Obrigado por me inspirarem a tentar sempre alcançar o meu melhor. Agradeço também aos meus irmãos, Rodrigo e Gabriela, que sempre acreditaram em mim e estiveram do meu lado, celebrando comigo cada conquista. Por fim, agradeço a todos os amigos que fiz ao longo do curso, que se tornaram parte fundamental da minha vida acadêmica e me ajudaram nos momentos mais difíceis. Sem vocês, essa jornada jamais teria sido tão prazerosa.

"Inteligência é a capacidade de evitar fazer o trabalho e, ainda assim, realizá-lo."
(Linus Torvalds)

Resumo

O mercado de queijos artesanais no Brasil apresenta um cenário promissor e em expansão, com uma receita aproximada de US$10 milhões em 2021 e uma projeção anual de crescimento de 2,04% até 2025, de acordo com o Sebrae - SC. Contudo, esse mercado enfrenta desafios consideráveis, como as rigorosas exigências sanitárias para os produtores e a falta de rastreabilidade na cadeia produtiva desse alimento. Para superar esse último obstáculo, propõe-se a implementação de uma rede blockchain utilizando a ferramenta Hyperledger Fabric. Esta solução possibilita a criação de um sistema de rastreabilidade seguro e transparente, permitindo o registro e verificação de todas as fases da cadeia de produção. Todas as partes interessadas dessa cadeia terão acesso à uma plataforma Progressive Web App (PWA), compatível com qualquer dispositivo, seja computador de mesa ou celular, facilitando o acompanhamento e registro de atividades. O consumidor final poderá, através de um código QR impresso na embalagem, acessar o histórico do queijo, garantindo a sua origem, autenticidade e segurança do produto. Ao longo deste trabalho, a solução proposta foi implementada e foram simulados alguns testes para garantir o funcionamento e desempenho do sistema proposto, abrindo caminho para futuras pesquisas e eventual implementação em condições reais de mercado.

Palavras-chave: Queijos Artesanais. Rastreabilidade. Cadeia de Produção. Blockchain. Hyperledger Fabric. Progressive Web App. PWA.

Abstract

The artisanal cheese market in Brazil presents a promising and growing scenario, with an estimated revenue of US$10 million in 2021 and an annual growth projection rate of 2.04% until 2025, according to Sebrae - SC. However, this market faces considerable challenges, such as the strict sanitary requirements for producers and the lack of traceability in the food production chain. To overcome this last obstacle, the implementation of a blockchain network using the Hyperledger Fabric platform is proposed. This solution allows for the creation of a secure and transparent traceability system, enabling the recording and verification of all stages of the production chain. All stakeholders in this supply chain will have access to a Progressive Web App (PWA) platform, compatible with any device, whether desktop or mobile, making it easier to track and monitor the activities. The end consumer will be able, through a QR code printed on the packaging, to access the history of the cheese, ensuring the product's origin, authenticity and safety. Throughout this work, the solution was implemented and several tests were simulated to ensure the system's functionality and performance, paving the way for future research and eventual implementation in real market conditions.

Keywords: Artisanal Cheeses. Traceability. Supply Chain. Blockchain. Hyperledger Fabric. Progressive Web App. PWA.

Lista de ilustrações

Lista de tabelas

Lista de siglas

API Application Programming Interface

CA Certificate Authority

GTIN Global Trade Item Number

JSON JavaScript Object Notation

MSP Membership Service Provider

MVCC MultiVersion Concurrency Control

MVP Minimum Viable Product

ORM Object Relational Mapper

PoS Proof of Stake

PoW Proof of Work

PWA Progressive Web App

QR Quick Response

REST Representational State Transfer

SDK Software Development Kit

SGTIN Serialised Global Trade Item Number

TCC Trabalho de Conclusão de Curso

URL Uniform Resource Locator

Sumário

Capítulo 1

Introdução

Um dos grandes desafios enfrentados pelas cadeias de suprimentos no Brasil, atualmente, é a dificuldade na rastreabilidade dos produtos, desde a produção até a distribuição para os consumidores finais. Rastreabilidade consiste na capacidade de obter acesso a toda e qualquer informação durante o ciclo de vida de um processo, através de identificações registradas (OLSEN; BORIT, 2013). A falta de uma solução adequada para esse problema gera diversas consequências significativas, como a distribuição de produtos falsificados, o que prejudica a segurança dos consumidores e a integridade do mercado. Esse problema foi observado em um estudo sobre a indústria farmacêutica nos Estados Unidos, que busca uma solução para a distribuição de medicamentos falsificados (UDDIN, 2021).

O mercado de queijos artesanais no Brasil gerou uma receita de US$9.929 milhões em 2021, com uma expectativa de crescimento de 2,04% ao ano até 2025 (SEBRAE, 2021). Apesar de promissor, esse mercado também sofre com a dificuldade de rastreabilidade dos produtos, o que é um problema crítico, especialmente por se tratar da indústria de alimentos. O método tradicional de rastreabilidade dos queijos artesanais no Brasil se baseia em registros manuais e sistemas de rastreamento pouco tecnológicos, sendo mais suscetíveis a erros e apresentando dificuldade na verificação dos dados. Uma possível solução é um sistema baseado em *blockchain*, similar ao proposto por Uddin.

A *blockchain*, criada em 2009 por um indivíduo ou grupo sob pseudônimo de Satoshi Nakamoto como parte da proposta para o Bitcoin (NAKAMOTO, 2009), é uma tecnologia inovadora que funciona como um livro-razão público e descentralizado, permitindo o registro de transações de forma segura e imutável. Uma vez que um bloco contendo um conjunto de transações é adicionado na cadeia, ele não pode ser alterado ou excluído, o que proporciona uma trilha de auditoria confiável. Essa natureza da *blockchain* faz com que ela seja ideal para aplicações de rastreabilidade, oferecendo um meio eficiente de

verificar a origem e jornada dos produtos, como proposto por Uddin.

Hyperledger Fabric é uma plataforma *blockchain* de código aberto projetada para uso empresarial (HYPERLEDGER FOUNDATION, 2024a). Uma das características que a torna interessante para o nosso problema é ser permissionada: os participantes dessa rede são conhecidos e verificados, e as permissões são atribuídas de acordo com seu cargo. Com isso em mente, propõe-se uma solução para o problema da rastreabilidade de queijos artesanais no Brasil através da criação de uma rede *blockchain* onde as etapas de produção, distribuição e venda dos queijos são registradas de forma segura e transparente. Isso permite que produtores, distribuidores, varejistas e consumidores tenham acesso a informações confiáveis sobre a origem e trajetória do produto, garantindo sua qualidade e autenticidade.

Essa rede *blockchain* poderá ser acessada por cada uma das quatro partes interessadas que serão cobertas neste trabalho: produtor, distribuidor, varejista e consumidor final. Seu acesso se dará por meio de uma plataforma PWA, através de um navegador em qualquer dispositivo. Os produtores, distribuidores e varejistas deverão se cadastrar na rede antes de utilizá-la. Feito o cadastro, cada um destes terá permissão de escrita, ficando responsável por registrar corretamente o que for solicitado, e de leitura, podendo conferir todas as etapas da cadeia. O consumidor final não terá registro na rede, tendo permissão apenas de leitura (*read-only*) das informações dos queijos.

A cadeia se inicia com o produtor registrando um produto e, a partir dele, podendo criar lotes deste queijo. Cada produto conta com uma série de etiquetas, cada uma com um código Quick Response (QR) único, que deverão ser impressas e coladas na embalagem de cada peça de queijo. O produtor pode escanear o código de um lote para ser redirecionado à página dele, onde poderá escanear códigos de embalagem, associando um ao outro. Essa associação gera uma peça de queijo, com as informações relevantes a produção.

As peças de queijo, por sua vez, podem ser transferidas de um produtor para um distribuidor. O distribuidor deve registrar o recebimento e também o momento em que o entregou para o varejista. O varejista deve registrar o recebimento do queijo e, posteriormente, sua venda. Por fim, o consumidor final, através da câmera do celular, poderá escanear o código QR colado na embalagem, sendo redirecionado para uma página mostrando todas as informações da peça de queijo e a trajetória do produto.

Capítulo 2

Objetivos

2.1 Objetivo Geral

O objetivo deste Trabalho de Conclusão de Curso (TCC) é a criação de um Minimum Viable Product (MVP) de uma rede *blockchain* para rastreabilidade de queijos artesanais utilizando Hyperledger Fabric, bem como uma plataforma PWA como *front-end* dessa aplicação.

2.2 Objetivos Específicos

1. Analisar e definir os requisitos do projeto, identificando as necessidades dos diversos atores da cadeia de produção dos queijos artesanais (*stakeholders*);
2. Elaborar o projeto de arquitetura, para desenvolver o modelo de blockchain adequado;
3. Construir o MVP, garantindo que ele atenda aos requisitos funcionais, de segurança e privacidade;
4. Avaliar o desempenho da rede em simular o rastreamento dos produtos.

2.3 Resultados Esperados

Espera-se que, ao fim deste trabalho, tenha-se obtido uma solução que englobe os seguintes pontos:

1. Melhoria na capacidade de rastrear a origem, a autenticidade e o histórico do produto ao longo da cadeia de valor dos queijos artesanais;

2. Proporcionar transparência nas transações relacionadas ao produto ao longo da execução dos processos na cadeia de valor dos queijos artesanais (incluindo produtores, distribuidores, varejistas e consumidores);

3. Aumento da confiança dos consumidores no produto, valorização da marca e melhoria da segurança alimentar.

Além disso, esse trabalho pode servir como referência para futuros projetos na mesma área. A ideia da rastreabilidade de cadeias de produção utilizando soluções baseadas em *blockchain* já é uma realidade, como veremos futuramente. É possível partir dos mesmos conceitos apresentados aqui para adaptar a solução para outros mercados.

Capítulo 3

Revisão Bibliográfica

3.1 Estado da Arte

Nesta seção, serão explorados os desenvolvimentos e pesquisas anteriores em relação a rastreabilidade em cadeias de produção, buscando identificar as principais descobertas e metodologias existentes na literatura, bem como as lacunas que ainda precisam ser preenchidas. O ponto de partida foi pesquisar artigos e trabalhos relacionados ao tema, como uma solução em Hyperledger Fabric criada para melhorar a rastreabilidade de medicamentos (UDDIN, 2021), sendo de grande ajuda como referência. O modelo de rastreabilidade usado por Uddin pode ser adaptado para os queijos artesanais, especialmente por já ter sido testado, e o artigo nos fornece importantes lições para abordagem do design e desenvolvimento do novo sistema.

Uddin usa como referência um artigo que nos fornece uma visão de como a tecnologia *blockchain* pode nos dar maior transparência no gerenciamento de uma cadeia produtiva (AZZI; CHAMOUN; SOKHN, 2019). Um ponto interessante desse artigo é a integração da *blockchain* com tecnologias já existentes que podem ajudar na rastreabilidade dos produtos, como sensores e códigos QR, sendo este último de fato usado em nossa solução.

Códigos QR são uma espécie de código de barras bidimensional, criado pela empresa Denso Wave, subsidiária da Toyota, em 1994. Eles são capazes de armazenar uma significativa quantidade de dados, desde texto até informações de contato e endereços Uniform Resource Locator (URL), graças ao seu formato quadrado, que permite a codificação de informações tanto na vertical quanto na horizontal (SOON, 2008). Esses códigos são facilmente escaneáveis por câmeras de celulares e outros dispositivos, tornando-os úteis para distribuição de informações de maneira compacta e rápida.

Na área de *blockchain* ligada à indústria alimentícia, temos um artigo que apresenta

o problema de segurança dos alimentos, destacando preocupações como a invisibilidade dos dados ao longo da cadeia de suprimentos e a possibilidade de alteração dos dados, presentes no sistema tradicional (LIN et al., 2020). Além disso, o artigo explica como a tecnologia *blockchain* é uma solução promissora, devido as suas características como irreversibilidade, contratos inteligentes e algoritmos de consenso.

Um outro trabalho relacionado a transparência propõe o uso de Hyperledger Fabric para aumentar a transparência dos cidadãos sobre as consultas de seus dados por entidades públicas (VALENTE; PAULINO, 2024). Uma prova de conceito é apresentada para mostrar a viabilidade do uso dessa plataforma, em que o histórico de consultas fica armazenado na rede *blockchain*, e o cidadão pode ver que organização fez a pesquisa e quais dados foram consultados.

Além dos artigos, foi importante buscar casos práticos de empresas que já utilizam Hyperledger Fabric em suas cadeias produtivas, para verificar se a realidade corresponde com a expectativa. Um estudo de caso mostra como o Walmart, maior varejista do mundo (DELOITTE, 2023), utilizou Hyperledger Fabric para implementar um sistema de rastreamento para produtos como carne de porco e mangas, reduzindo o tempo necessário para rastreá-los de dias para segundos (WALMART, 2019). Este sistema permitiu maior segurança e transparência no processo, provando que a tecnologia *blockchain* pode ser utilizada para melhoria da rastreabilidade nas indústrias.

3.2 Identificação dos Produtos

Para identificação das peças de queijo de maneira similar às condições reais de mercado, utilizaremos como base números do padrão GS1 conhecidos como Global Trade Item Number (GTIN) e Serialised Global Trade Item Number (SGTIN). O GTIN identifica um grupo de produtos idênticos, sendo formado pelo prefixo da empresa que o produziu e o número de referência desse produto. O SGTIN, por sua vez, adiciona ao final do GTIN a parte serial do item, sendo capaz de identificar cada peça individualmente (VEVLE et al., 2018).

O uso do SGTIN é discutido no contexto de identificação de produtos alimentícios e rastreabilidade na cadeia de produção, como forma de trazer melhorias para a logística. Os autores também citam o uso de códigos QR em substituição aos tradicionais códigos de barra (SCHOLTEN et al., 2016). No nosso caso, a identificação das peças de queijo será feita com um número em formato similar ao SGTIN-96 (binário com 96 bits), facilitando uma possível transição para o padrão GS1 no futuro, em condições reais de mercado.

Cada produtor que se registrar na rede receberá um código, sendo este um número binário de 9 bits, sequencial de acordo com a ordem de registro. Ao registrar um queijo, este produto será identificado com um GTIN que combina os 9 bits do código do produtor com 11 bits do código deste queijo específico, também de maneira sequencial. Por fim,

quando peças de queijo forem geradas a partir deste produto, haverá a combinação deste GTIN com um número sequencial de 76 bits, que identifica a ordem da criação das peças.

Esse SGTIN gerado será gravado internamente, mas antes de ser mostrado ao usuário, ele será convertido para um número decimal de 13 dígitos, dividido em três partes separadas por pontos, com três dígitos para o código do produtor, cinco dígitos para o código do produto e cinco dígitos para a peça. O motivo disso é que SGTINs são números binários muito longos, que funcionam bem para uso no sistema, mas são pouco atrativos para os usuários. Assim que um produto é registrado pelo usuário, são geradas 50 etiquetas com SGTINs sequenciais. Sendo assim, o primeiro produtor registrado na rede, ao registrar seu primeiro produto, poderá ver que este possui, inicialmente, as etiquetas de número **001.00001.00001** até **001.00001.00050**.

3.3 Blockchain

Blockchain é uma tecnologia de banco de dados distribuído que permite a criação de uma cadeia de blocos, como o próprio nome sugere. Esses blocos armazenam um conjunto de transações, e não podem ser modificados ou deletados sem consenso da rede, garantindo um histórico transparente e confiável (TRIPATHI; AHAD; CASALINO, 2023). Cada bloco na cadeia pode ser identificado por um *hash*, que é uma sequência de caracteres de tamanho fixo gerado a partir da criptografia de uma mensagem original de qualquer tamanho (PIERRO, 2017).

O primeiro bloco da cadeia é chamado de bloco gênese (*genesis block*). Um bloco é composto por seu *hash*, um carimbo de tempo (ou *timestamp*), o *hash* do bloco anterior (no caso do bloco gênese, um ponteiro nulo) e um número de transações. A Figura 1 mostra um conjunto de blocos em uma *blockchain*.

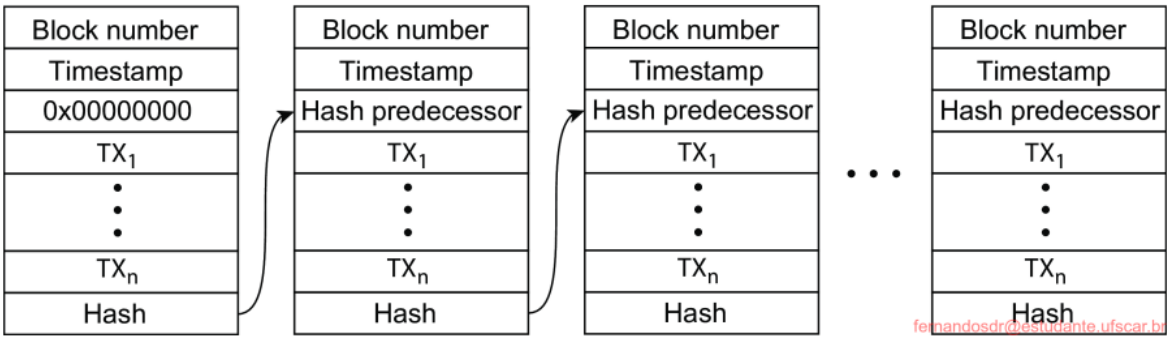

Figura 1 – Histórico de transações encadeadas em blocos (*blockchain*) - Fonte: (STEEN; TANENBAUM, 2023)

Cada participante de uma rede *blockchain* é chamado de nó (ou *node*). Uma das vantagens da natureza distribuída da rede *blockchain* é não depender de uma entidade

centralizada confiável: cada nó, a princípio, não é confiável, existindo um algoritmo de consenso para definir se uma transação deve ser realizada. Todos os participantes da rede podem observar tudo que aconteceu desde o começo e, consequentemente, verificar a validade de uma transação (STEEN; TANENBAUM, 2023).

Uma rede *blockchain* pode ser pública, privada ou consórcio. Redes públicas são abertas a qualquer usuário que queira participar da rede, permitindo que este submeta transações e participe do processo de consenso. As privadas são restritas a participantes específicos, ou seja, apenas entidades selecionadas podem participar da rede, e as permissões de escrita são centralizadas em uma organização. Já o consórcio combina características das duas anteriores, onde o processo de consenso é controlado por um grupo pré-selecionado de nós, e o direito de leitura pode ser público ou restrito aos participantes (BUTERIN, 2015), como no caso desse projeto, em que temos produtores, distribuidores e varejistas já conhecidos.

Algoritmos de consenso são protocolos que buscam resolver o problema de confiança em uma rede envolvendo múltiplos nós não confiáveis (BACH; MIHALJEVIC; ZAGAR, 2018). Nas *blockchains* públicas, um algoritmo comum é o *Proof of Work (PoW)*, usado pelo Bitcoin, que impõe um desafio criptográfico complexo e que requer um poder computacional significativo para ser decifrado pelos nós. O primeiro nó que consegue resolvê-lo ganha o direito de adicionar um novo bloco na cadeia, processo chamado de mineração (LASHKARI; MUSILEK, 2021). A ideia é fazer com que, por meio dos desafios, não seja lucrativo para os agentes maliciosos agirem na rede, já que isso pode resultar em recursos desperdiçados sem recompensa.

Um dos problemas apontados para o PoW é o consumo excessivo de energia, visto que os nós que não conseguiram resolver o desafio a tempo gastaram energia em vão, pois apenas o primeiro é recompensado. Como alternativa, no campo das cripto-moedas, foi criado o *Proof of Stake (PoS)*, em que os nós são selecionados para adicionar um novo bloco na cadeia com base na quantidade de moedas que possuem, partindo do princípio que usuários que possuem muitas moedas são os mais interessados em manter a rede segura (ASIF; HASSAN, 2023).

As *blockchains* privada e consórcio não apresentam necessidade de algoritmos de consenso tão complexos, pois a participação na rede é mais restrita. Como os nós participantes são conhecidos, um algoritmo mais simples já é suficiente para evitar que transações maliciosas sejam realizadas. No caso de Hyperledger Fabric, o algoritmo de consenso padrão é o Raft, que é um consenso por votação: nós candidatos recebem votos de nós seguidores, sendo o líder eleito pela maioria e responsável por receber as requisições do cliente, acrescentar os comandos ao seu registro, e replicar este registro aos nós seguidores. Quando a maioria dos seguidores confirma o recebimento, o líder registra as alterações e informa-os para que façam o mesmo (HUANG; MA; ZHANG, 2020).

3.4 Hyperledger Fabric

Hyperledger é um projeto colaborativo de *blockchain* para empresas iniciado em 2015 pela Linux Foundation, que fornece diversos *frameworks* de código aberto para diferentes funções (LINUX FOUNDATION, 2024). Em 2016, foi lançada a plataforma Hyperledger Fabric, que foi a primeira ferramenta escolhida para implementação do nosso sistema. Os motivos para sua escolha incluem ser uma *blockchain* permissionada e ser projetada para uso comercial, o que faz com que consiga lidar com grandes volumes de transação de forma eficiente. Além disso, como citado anteriormente, já existem grandes empresas no mercado utilizando essa ferramenta, o que garante que já está validada.

Os contratos inteligentes (*chaincodes*) da rede oficialmente suportam código escrito em Go, Java e JavaScript ou TypeScript, por meio de um Software Development Kit (SDK) fornecido para cada linguagem. Este é um diferencial dessa ferramenta, já que antes disso, era necessário utilizar linguagens específicas para escrever contratos inteligentes, como Solidity para a rede Ethereum. As próximas subseções irão introduzir conceitos importantes para compreender o funcionamento da rede que será criada.

3.4.1 Livro Razão (*Ledger*)

Um *ledger*, ou livro razão, é um registro imutável de todas as transações que ocorreram na rede *blockchain*. Ele é formado por duas partes: o *world state*, que é um banco de dados contendo o estado atual dos ativos (chamados de *assets*), facilitando para que o programa acesse o valor atual do estado, e a *blockchain*, que é a cadeia de blocos contendo todos os registros das transações que resultaram no *world state*.

3.4.2 Contrato Inteligente (*Chaincode*)

Chaincode é a nomenclatura utilizada pelo Hyperledger Fabric para o programa que contém os contratos inteligentes. O *chaincode* define quais são os ativos e as instruções para que estes sejam criados, atualizados ou deletados, além de consultar dados no livro razão (chamado de *ledger*). Os participantes da rede invocam o *chaincode* por meio de propostas de transações, que fazem com que ele interaja com o *ledger*, garantindo a autenticidade e registro destas na *blockchain*.

3.4.3 Peers

Peers, ou *peer nodes*, são os nós da rede *blockchain* que mantêm uma cópia do *ledger* e executam o *chaincode*, sendo responsáveis por validar e endossar as transações propostas pelo cliente, garantindo que as instruções sejam seguidas. Quando a transação enviada pelo cliente é endossada pelos *peers*, esta é encaminhada para o serviço de ordenação

(*orderers*). Depois da ordenação, cada *peer* submete o bloco ordenado de transações na *ledger*, atualizando seu estado atual com o resultado das transações válidas.

3.4.4 Ordenadores (*Orderers*)

Os *orderers*, ou nós ordenadores, são componentes do sistema responsáveis por ordenar as transações em uma sequência consistente. Depois que as transações são validadas pelos *peers*, elas são enviadas para os ordenadores, que as agrupam em blocos e seguem um algoritmo de consenso para garantir que todos os nós concordam sobre a ordem da transação. Então, os ordenadores criam um bloco ordenado contendo essas transações e distribuem esse bloco de volta para os *peers*. Os nós ordenadores podem formar uma organização própria ou pertencer a uma outra organização.

3.4.5 Autoridades Certificadoras

Cada participante da rede, como *peers*, *orderers* e administradores, possui uma identidade digital encapsulada como um certificado digital X.509. Esses certificados são essenciais pois determinam as permissões que cada ator da *blockchain* possui sobre os recursos e informações da rede, sendo emitidos e gerenciados por uma *Certificate Authority (CA)*.

3.4.6 Gestor de Associação (*Membership Service Provider*)

Como explicado anteriormente, os CAs emitem certificados digitais para que cada ator possa provar sua identidade. Para que essa identidade seja reconhecida pela rede, precisamos de um *Membership Service Provider (MSP)*, que irá verificar e autenticar cada participante, por exemplo, verificando se um determinado *peer* pode endossar uma transação. Dessa forma, o MSP torna uma identidade em um papel (chamado de *role*), permitindo que um ator exerça suas funções na rede.

3.4.7 Organizações

Organizações, também conhecidas como "membros", são entidades administrativas que agrupam nós, ou *peers*, e colaboram e interagem entre si dentro da rede *blockchain*, através dos canais.

3.4.8 Canais

Canais são sub-redes privadas dentro de uma *blockchain* principal. Cada canal possui seu próprio conjunto de *peers*, seu *chaincode* e seu *ledger*, garantindo que apenas as organizações pertencentes a ele podem ver e submeter transações, proporcionando um alto nível de privacidade para o sistema.

Capítulo 4

Desenvolvimento

4.1 Análise de Requisitos

Para a análise de requisitos, precisamos entender as funcionalidades e serviços necessários para que tenhamos um MVP do nosso produto. As seguintes ações deverão poder ser realizadas pelos usuários:

1. Registro dos usuários no sistema (produtores, distribuidores e varejistas);
2. Registro de novos produtos pelos produtores;
3. Registro de um lote de queijo a partir de um produto;
4. Registro de peças de queijo através das etiquetas, em um determinado lote;
5. Registro das transferências das peças de queijo entre produtores, distribuidores e varejistas;
6. Registro da venda de uma peça de queijo pelo varejista;
7. Consulta ao histórico das peças de queijo.

Dessa forma, será necessário desenvolver um *front-end* com telas para navegação dos usuários, um serviço *back-end* para fazer as chamadas de funções do *chaincode*, uma Application Programming Interface (API) para integração entre *front-end* e *back-end*, e a rede *blockchain* em Hyperledger Fabric. Os diagramas apresentados abaixo nos ajudam a entender o funcionamento do sistema.

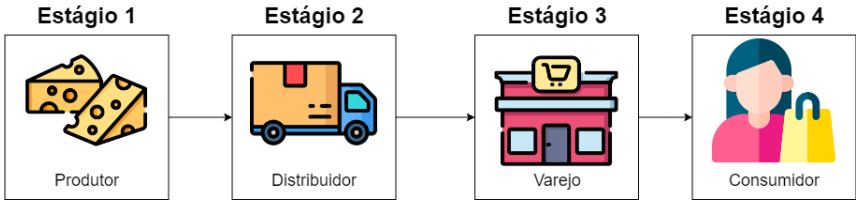

Figura 2 – Cadeia de abastecimento de queijos artesanais - Fonte: Autor, (Freepik, 2024)

Na Figura 2, podemos observar uma simplificação do modelo tradicional da cadeia de produção de queijos artesanais. O problema deste modelo é que a informação que o consumidor tem sobre o processo é praticamente nula, pois não sabe nada sobre o produtor ou distribuidor. A ideia proposta é que o fluxo de informações durante todo o processo esteja disponível por todas as partes interessadas, aumentando a transparência da cadeia.

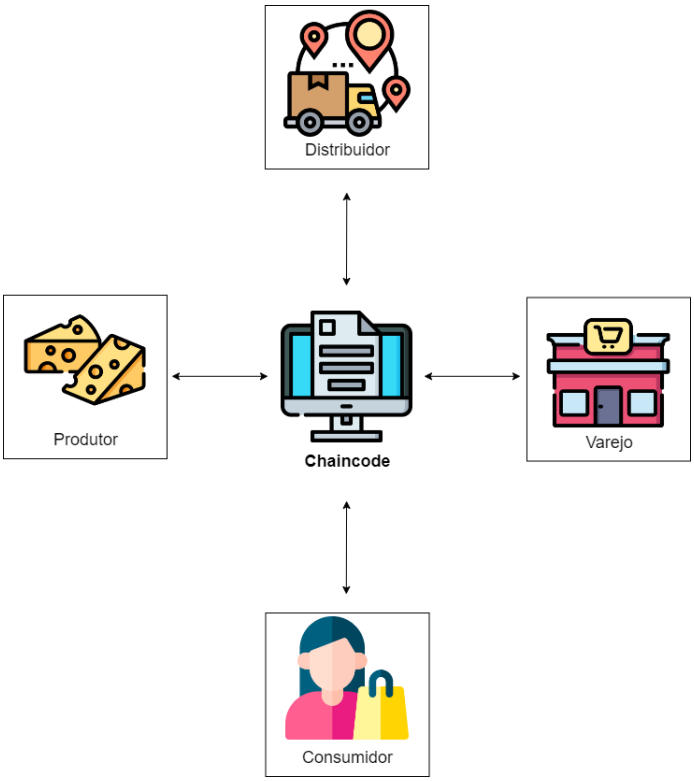

Figura 3 – Diagrama de modelo proposto - Fonte: Autor, (Freepik, 2024)

No modelo visto na Figura 3, todas as partes se comunicam através do *chaincode* presente no sistema, garantindo a transparência e visibilidade de todo o processo para todas as partes interessadas. Com isso em mente, podemos criar um diagrama de fluxo mais detalhado, destacando as partes técnicas do sistema, como pode ser visto na Figura 4.

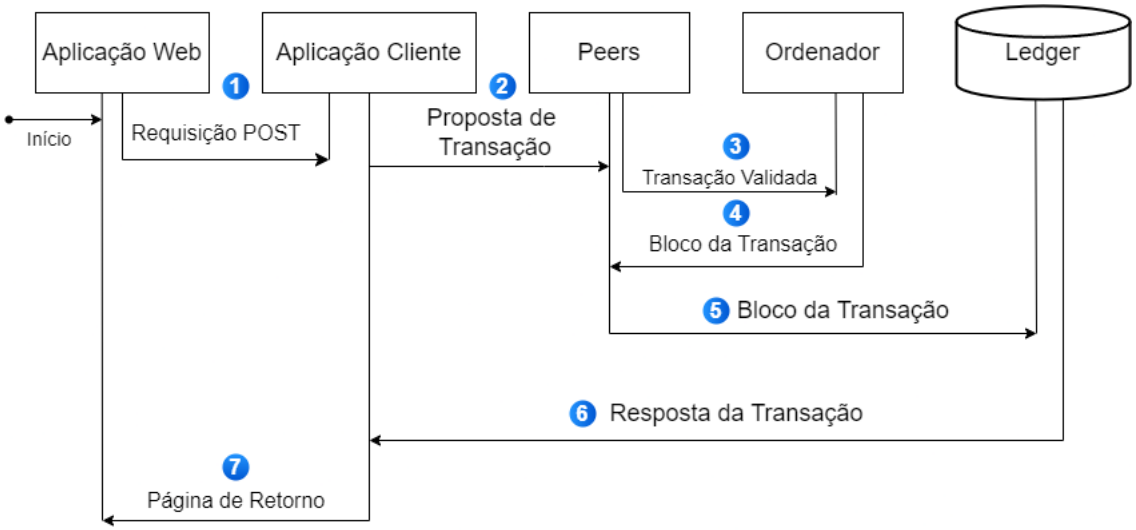

Figura 4 – Diagrama de fluxo do sistema - Fonte: Autor

O fluxo começa com o usuário interagindo com a aplicação web, gerando uma requisição para a aplicação cliente, que por sua vez irá mandar uma proposta de transação para os *peers*. Os *peers* simulam a transação para verificar se são válidas e, caso sejam, as mandam para o ordenador, que as agrupa em blocos e faz a ordenação. Esse bloco ordenado é devolvido para os *peers*, que irão validar cada transação para garantir que não houveram inconsistências desde a simulação. Caso a transação seja validada, ela é aplicada ao *ledger*, onde serão escritas as modificações propostas, etapa chamada de *commit*. Uma vez que a transação for confirmada, uma resposta de sucesso é enviada ao cliente, que irá redirecionar o usuário até a página de retorno da requisição dele.

A partir dessa explicação, podemos começar a pensar em cada um dos elementos da arquitetura isoladamente.

4.2 Desenvolvimento do MVP

4.2.1 Front-end

Para o *front-end* da aplicação, é necessário entender que nosso sistema deverá funcionar de maneira eficiente tanto em computadores (*desktops* ou *notebooks*) quanto em dispositivos móveis (como celulares e *tablets*). Uma ferramenta encontrada para esse fim foi o PWA, que funciona em qualquer dispositivo com um navegador, o que elimina a necessidade de desenvolver um aplicativo diferente para cada sistema operacional. Dessa forma, o usuário não precisa instalar um aplicativo específico e suas atualizações, visto que quando o desenvolvedor fizer um novo *deploy* de atualização, basta o usuário se conectar novamente à página para usufruir dela.

O *front-end* apresenta telas para navegação e utilização da aplicação pelo usuário, desenvolvidas na linguagem TypeScript, definida pela Microsoft como um JavaScript que inclui sintaxe para tipos (MICROSOFT, 2023), e com o *framework* Next.js, que permite a criação de aplicações web de alta qualidade com o poder dos componentes React (NEXT.JS, 2024). Este *framework* facilita o gerenciamento das rotas a partir da estrutura dos arquivos do projeto: adicionando um arquivo "page.tsx" dentro de uma pasta, automaticamente existirá uma página com o mesmo nome da pasta, na mesma estrutura. Isso evita o trabalho de ter que rotear cada uma das páginas para a URL desejada, automatizando o processo.

Outra vantagem do Next.js é a facilidade de se controlar acessos em determinadas páginas, podendo protegê-las editando o arquivo "middleware.ts". Ele foi configurado de forma que, quando não há uma sessão ativa, as únicas páginas que podem ser acessadas são a de *login*, a de registro, e a de informações de um queijo; qualquer outra página irá redirecionar o usuário à página de *login*. De maneira similar, usuários com uma sessão ativas que tentarem acessar a página de *login* ou de registro serão redirecionados automaticamente ao seu painel (*dashboard*).

Next.js também conta com bibliotecas que nos ajudam em outros aspectos do projeto. Por exemplo, para a parte de autenticação foi usada a biblioteca "next-auth" (NEXTAUTH.JS, 2024), que facilita o registro e login dos usuários de maneira totalmente segura. Além disso, a biblioteca "next-pwa" (NEXT-PWA, 2024) permitiu a criação do PWA de maneira extremamente simples, bastando seguir alguns poucos passos como atualizar o arquivo "next.config.js" e criar o "manifest.json".

Escolhidas as tecnologias a serem utilizadas no *front-end*, foram criadas as páginas necessárias para o nosso MVP. A seguir, serão mostradas as capturas de tela das páginas desenvolvidas, tendo como foco principal seu uso em dispositivos móveis. A tela inicial, mostrada na Figura 5, é a de *login*, onde o usuário acessa sua conta com seu e-mail e senha.

Figura 5 – Tela de *login* - Fonte: Autor

Caso o usuário não esteja cadastrado, ele pode clicar em "Cadastre-se" para ser redirecionado à página de cadastro, mostrada na Figura 6.

Figura 6 – Tela de registro - Fonte: Autor

Nessa tela, o usuário deve escolher seu cargo entre as opções "Produtor", "Distribuidor" e "Varejista", além de preencher suas informações como nome, CNPJ, e-mail e definir sua senha. Após o cadastro, ele será redirecionado de volta para a página de *login*, podendo dessa vez acessar o sistema. Ao fazer isso, o usuário irá se deparar com um *dashboard* de acordo com sua função. Produtores acessarão o *dashboard* de produtor, mostrado na Figura 7, onde terão as opções de registrar um novo produto, registrar um novo lote, ver suas etiquetas, ver seus lotes e ver seus queijos.

Figura 7 – Tela de *dashboard* do produtor - Fonte: Autor

Ao clicar em "Registrar Novo Produto", o produtor deverá inserir o nome de seu produto para registro, como pode ser visto na Figura 8. Após clicar em "Registrar", serão automaticamente geradas 50 etiquetas para este produto.

Figura 8 – Tela de registro de produto - Fonte: Autor

Acessar "Minhas Etiquetas" no *dashboard* levará o usuário para uma página em que ele poderá selecionar um de seus produtos, como mostrado na Figura 9.

Figura 9 – Tela de seleção de produto para ver etiquetas - Fonte: Autor

Cada produto criado possui um GTIN para identificação, formado pelo ID do produtor e ID do produto, separados por ponto. Selecionar um desses GTINs irá levar o usuário para a página da Figura 10, onde ele poderá ver e imprimir as etiquetas desse produto.

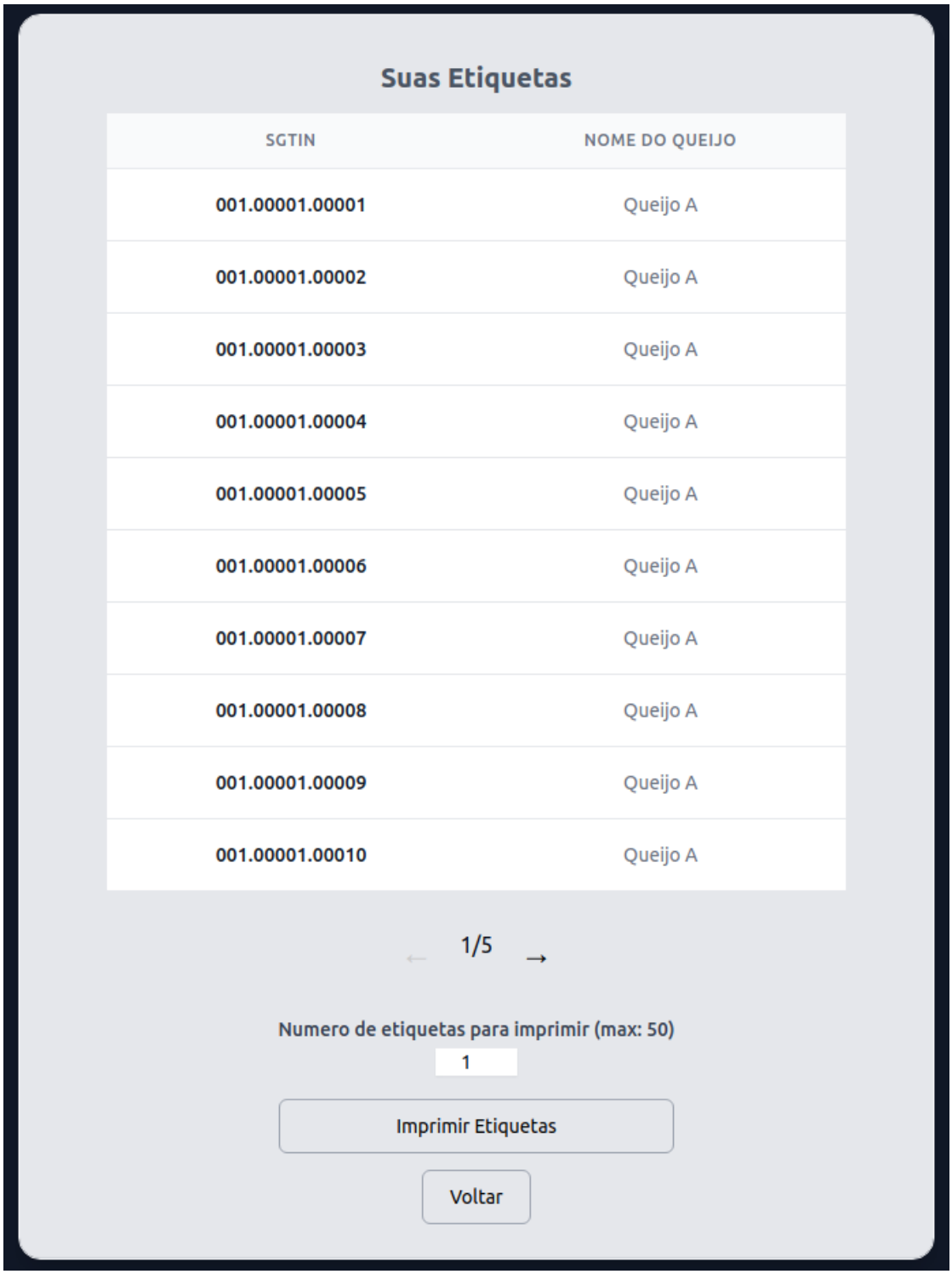

Figura 10 – Tela de etiquetas - Fonte: Autor

Etiquetas, por sua vez, possuem um SGTIN para identificação. Além do ID do pro-

dutor e do produto, elas também contam com um número serial, que será diferente para cada peça de queijo. O usuário pode selecionar a quantidade de etiquetas para impressão e clicar em "Imprimir Etiquetas", o que fará a tela de impressão do navegador abrir com o número de etiquetas solicitadas, como mostrado na Figura 11.

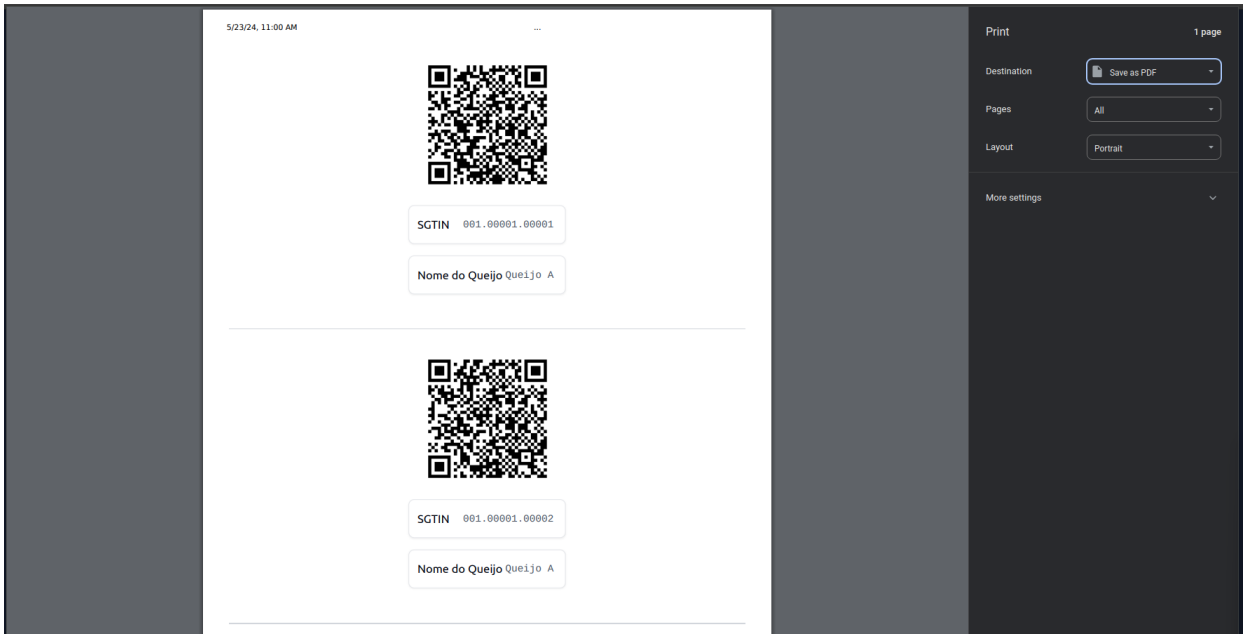

Figura 11 – Tela de impressão contendo duas etiquetas - Fonte: Autor

Para que o usuário registre uma peça de queijo, ele precisa antes criar um lote do produto. Isso pode ser feito a partir da opção "Registrar Novo Lote", no *dashboard*. Ao entrar na página, a utilização da localização do usuário será solicitada, como mostra a Figura 12, e deverá ser permitida.

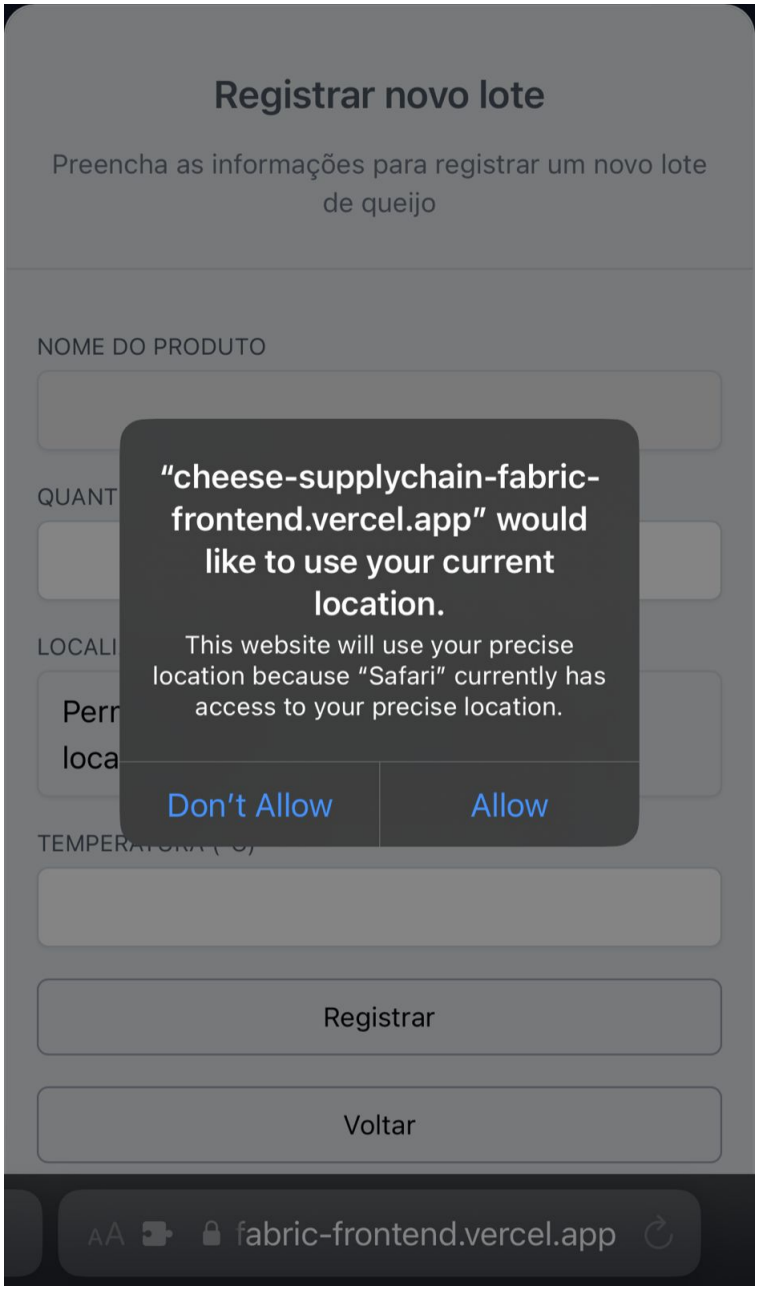

Figura 12 – Solicitação de permissão da localização - Fonte: Autor

Nessa tela, mostrada na Figura 13, o produtor deverá preencher as informações do lote, selecionando o produto desejado e inserindo a quantidade, em litros, e a temperatura. Após isso, o usuário deve clicar em "Registrar" para o novo lote ser gerado.

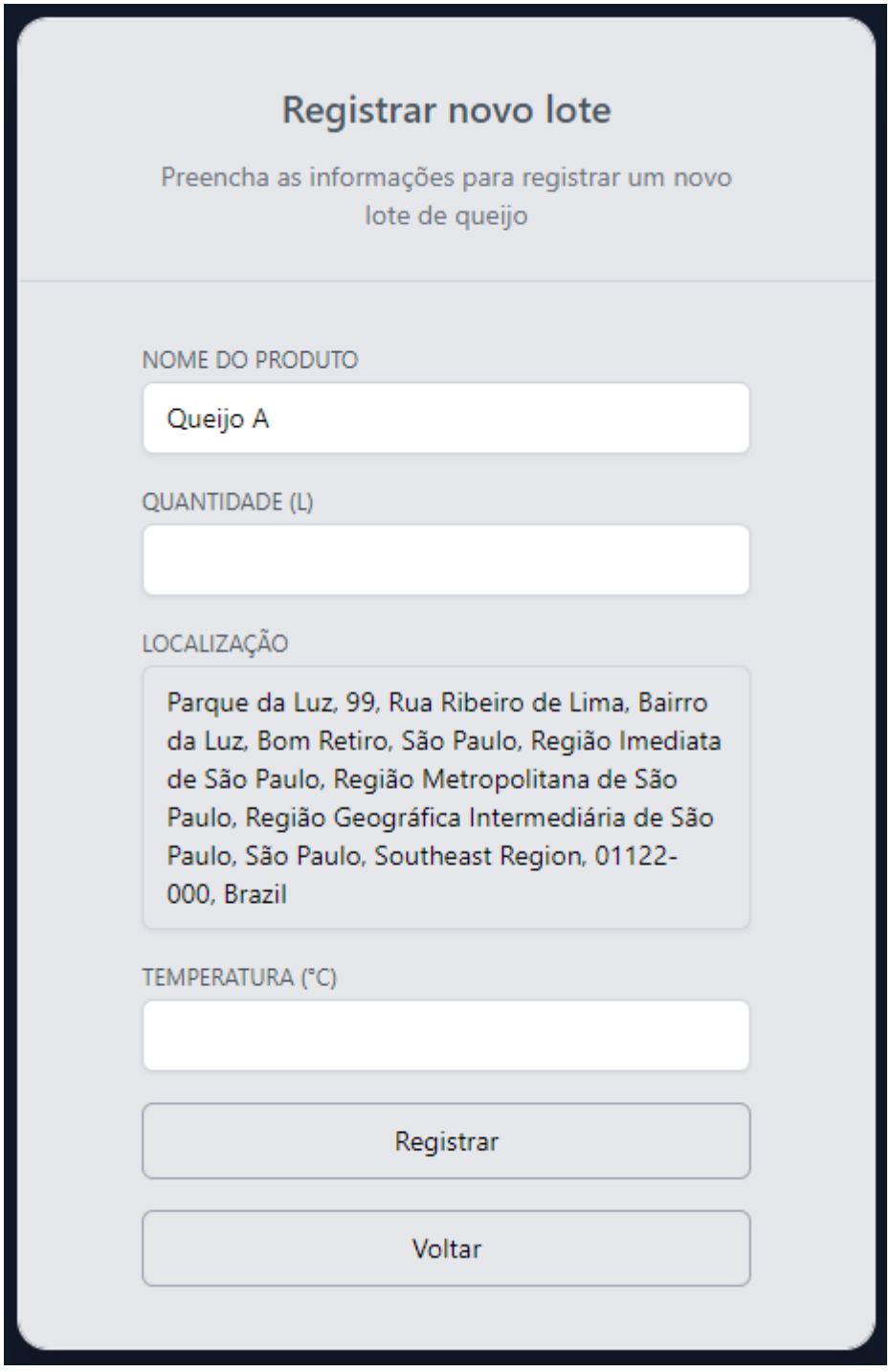

Figura 13 – Tela de registro de lote de queijo - Fonte: Autor

Após o registro do novo lote de queijo, o produtor será redirecionado para uma tela onde poderá ver todos os lotes que pertencem a ele. Essa tela, que pode ser vista na Figura 14, também é acessível a partir do *dashboard*, selecionando a opção "Meus Lotes".

Figura 14 – Tela contendo todos os lotes pertencentes ao produtor - Fonte: Autor

Clicar em um dos IDs mostrados nessa tela levará o usuário a uma página contendo todas as informações do lote de queijo em questão, com um código QR gerado a partir de seu ID e que redireciona para essa mesma página. A Figura 15 mostra a tela de detalhes de um lote de queijo registrado por um produtor.

Figura 15 – Tela contendo detalhes do lote de queijo - Fonte: Autor

Como é possível observar, além de também possuir um botão de impressão, similar ao das etiquetas, existe também o botão "Escanear etiqueta". Clicar nesse botão irá abrir uma tela e solicitar ao usuário a permissão do uso da câmera, como mostra a Figura 16.

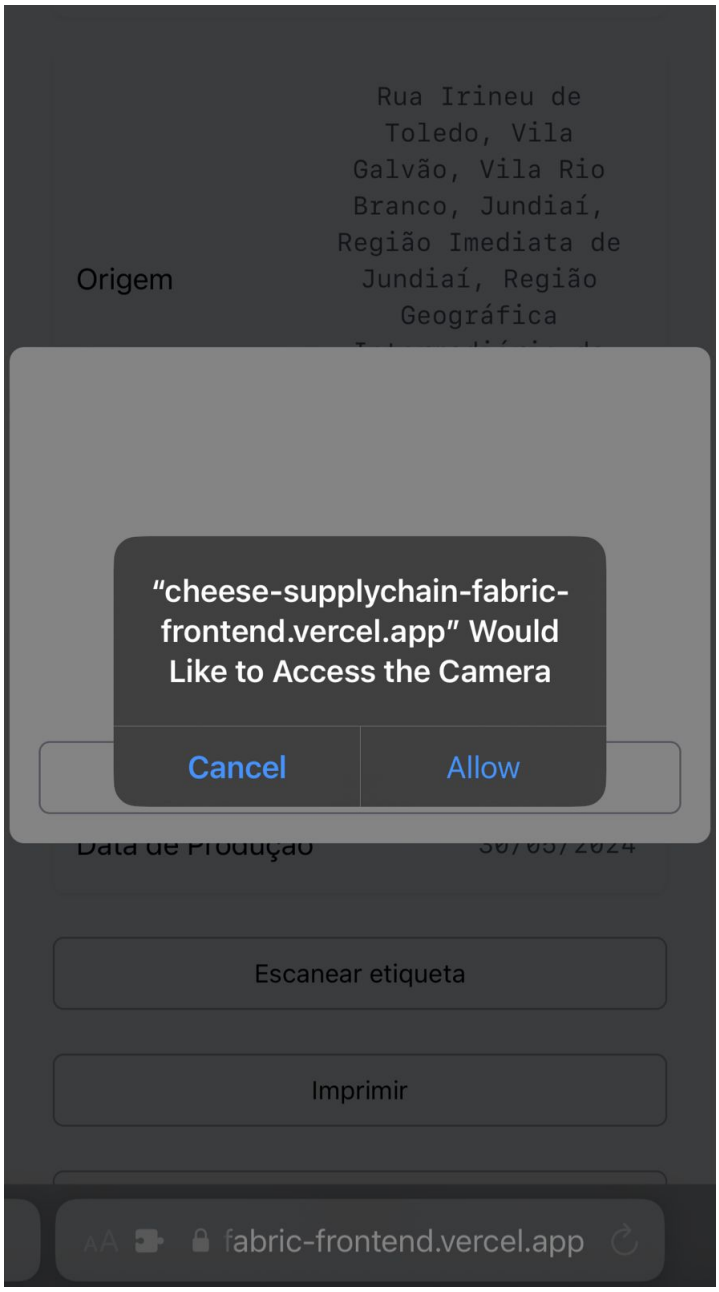

Figura 16 – Solicitação do uso de câmera - Fonte: Autor

Após o uso da câmera ser permitido, o usuário deverá apontar a câmera para o código QR da etiqueta, como pode ser visto na Figura 17.

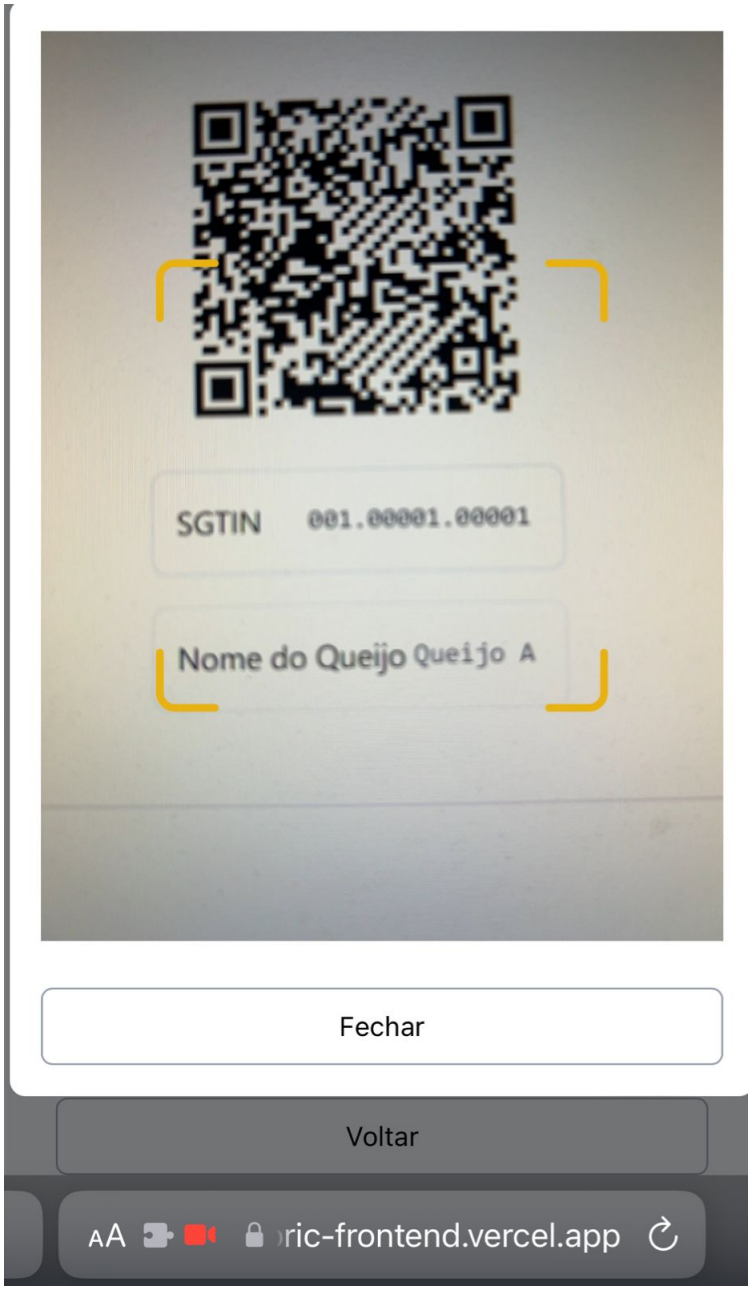

Figura 17 – Tela de câmera com um código QR sendo apontado - Fonte: Autor

Uma vez que o código QR for escaneado, será feita a associação entre a etiqueta e o lote. Se esse processo for bem sucedido, a tela de câmera fechará e o usuário receberá uma mensagem de sucesso, que também indica o SGTIN do queijo criado, como mostrado na Figura 18.

Figura 18 – Mensagem de sucesso ao registrar um queijo - Fonte: Autor

O usuário pode repetir o processo mencionado para criar quantos queijos quiser, a partir das etiquetas criadas. Quando a última etiqueta for utilizada, 50 novas serão criadas automaticamente. A partir do *dashboard*, o produtor poderá clicar em "Meus Queijos" para ver todos os queijos criados por ele, levando-o a uma página parecida com

a mostrada na Figura 19.

Figura 19 – Tela contendo os queijos - Fonte: Autor

Clicar em um SGTIN irá redirecionar o usuário para uma página similar à mostrada na Figura 20, com os detalhes do queijo.

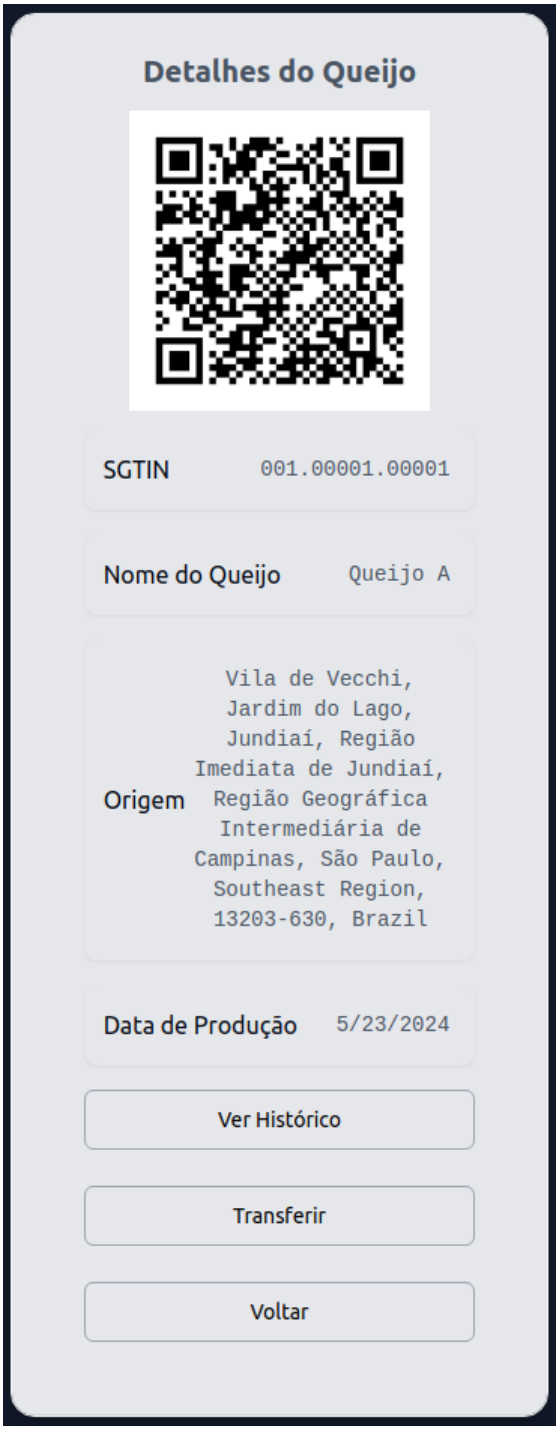

Figura 20 – Tela de detalhes do queijo - Fonte: Autor

Além das informações deste queijo em específico, como sua origem e data de produção, é possível observar o histórico clicando em "Ver Histórico". Como pode-se observar na Figura 21, o queijo recém criado possui apenas a informação de "Produzido", com data e horário do registro.

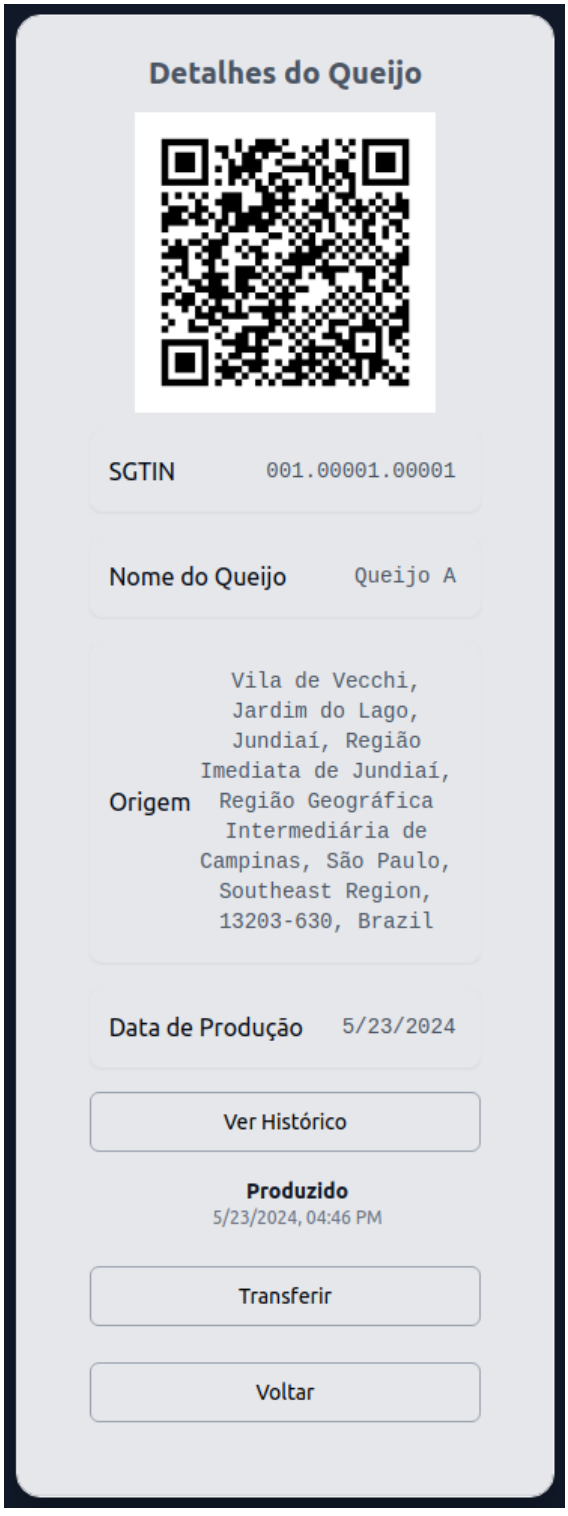

Figura 21 – Tela de lote de queijo após transferência para o varejista - Fonte: Autor

O produtor também poderá transferir o queijo para um distribuidor, clicando no botão "Transferir". Isso irá abrir a tela de transferência, mostrada na Figura 22, em que o produtor deve digitar o e-mail do distribuidor que levará o queijo a um varejista e clicar em "Enviar".

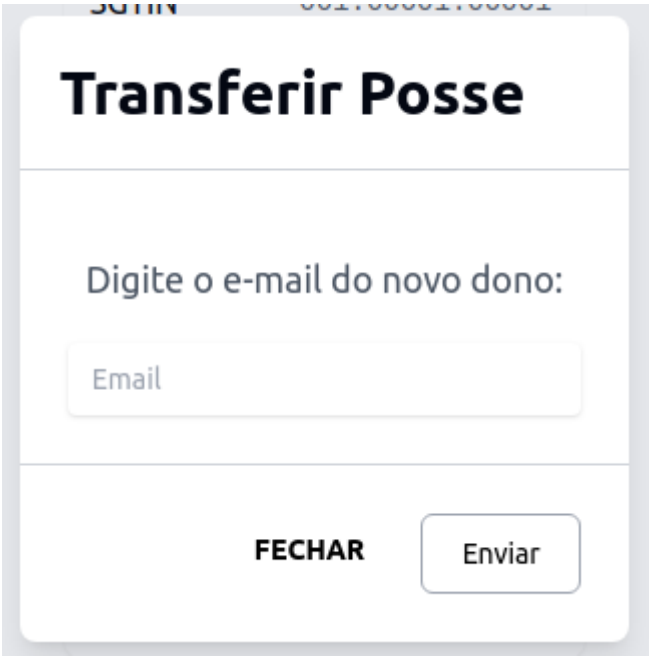

Figura 22 – Tela de transferência de queijo - Fonte: Autor

Queijos que foram transferidos passarão a ter o status "Pendente", que pode ser visto na página de queijos, mostrada na Figura 23. Esse estado permanecerá até que o distribuidor confirme o recebimento.

Figura 23 – Tela de queijos com status Pendente - Fonte: Autor

O *dashboard* do distribuidor possui apenas a opção de ver queijos, como mostra a Figura 24.

Figura 24 – Tela de dashboard do distribuidor - Fonte: Autor

Clicar em "Meus Queijos" levará o distribuidor até a tela da Figura 25, que inclui todos os queijos que estão esperando a confirmação dele, os que estão em posse dele, e os que já estiveram em posse dele.

Figura 25 – Tela de queijos do distribuidor - Fonte: Autor

Ao selecionar o queijo que foi transferido para ele, o distribuidor verá a opção de "Confirmar Transferência" junto aos detalhes do queijo, como pode ser visto na Figura 26.

Figura 26 – Tela do queijo aguardando confirmação do distribuidor - Fonte: Autor

Clicar no botão irá fazer com que o distribuidor confirme o recebimento do queijo e obtenha sua posse. O histórico será atualizado com o status atual "Enviado", e o botão de "Transferir"agora aparece para este usuário, como mostra a Figura 27.

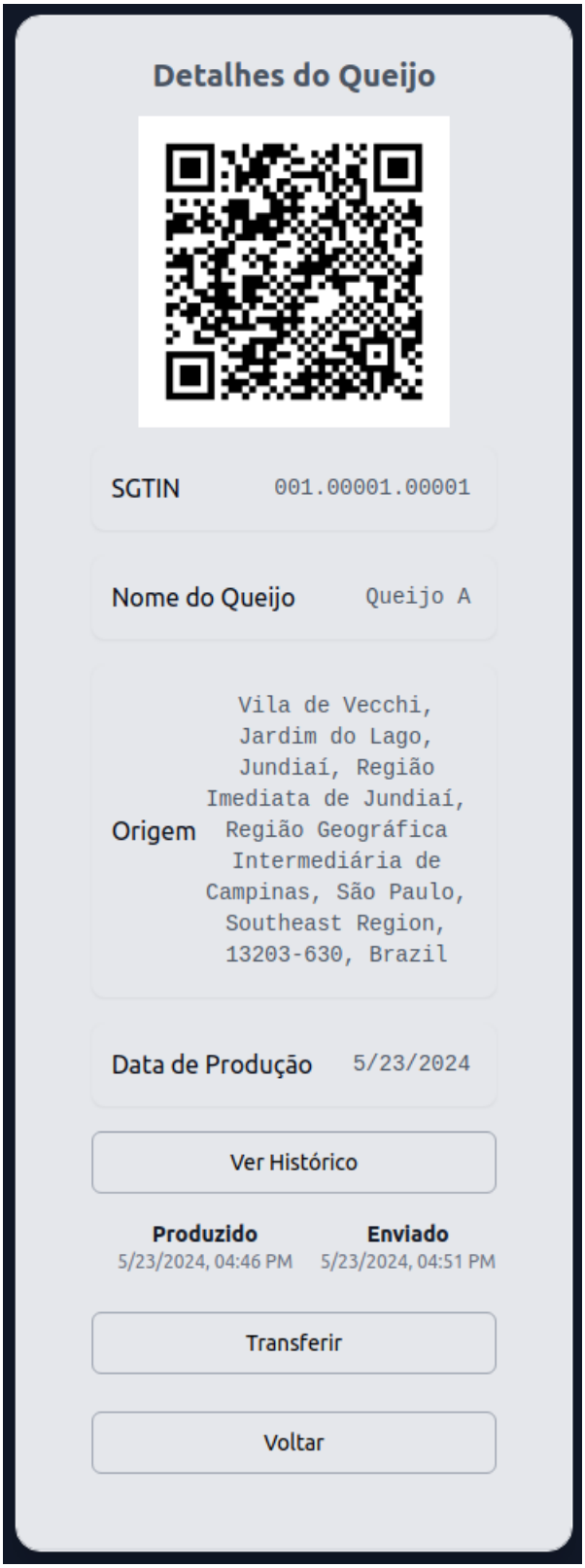

Figura 27 – Tela de detalhes do queijo após confirmado pelo distribuidor - Fonte: Autor

O distribuidor deve clicar em "Transferir"para registrar que entregou o queijo ao varejista, na mesma tela da Figura 22, exatamente como feito pelo produtor. Com isso feito, o

queijo retorna ao estado "Pendente", aguardando a confirmação do varejista. O *dashboard* do varejista, visto na Figura 28, é bem semelhante ao do distribuidor, com apenas a opção "Meus Queijos".

Figura 28 – Tela de dashboard do varejista - Fonte: Autor

Uma página similar à da Figura 25 aparecerá para o varejista ao clicar nessa opção. Selecionar o queijo levará o varejista para uma página com o mesmo botão de "Confirmar Transferência", porém ele encontrará o histórico "Enviado", como mostra a Figura 29.

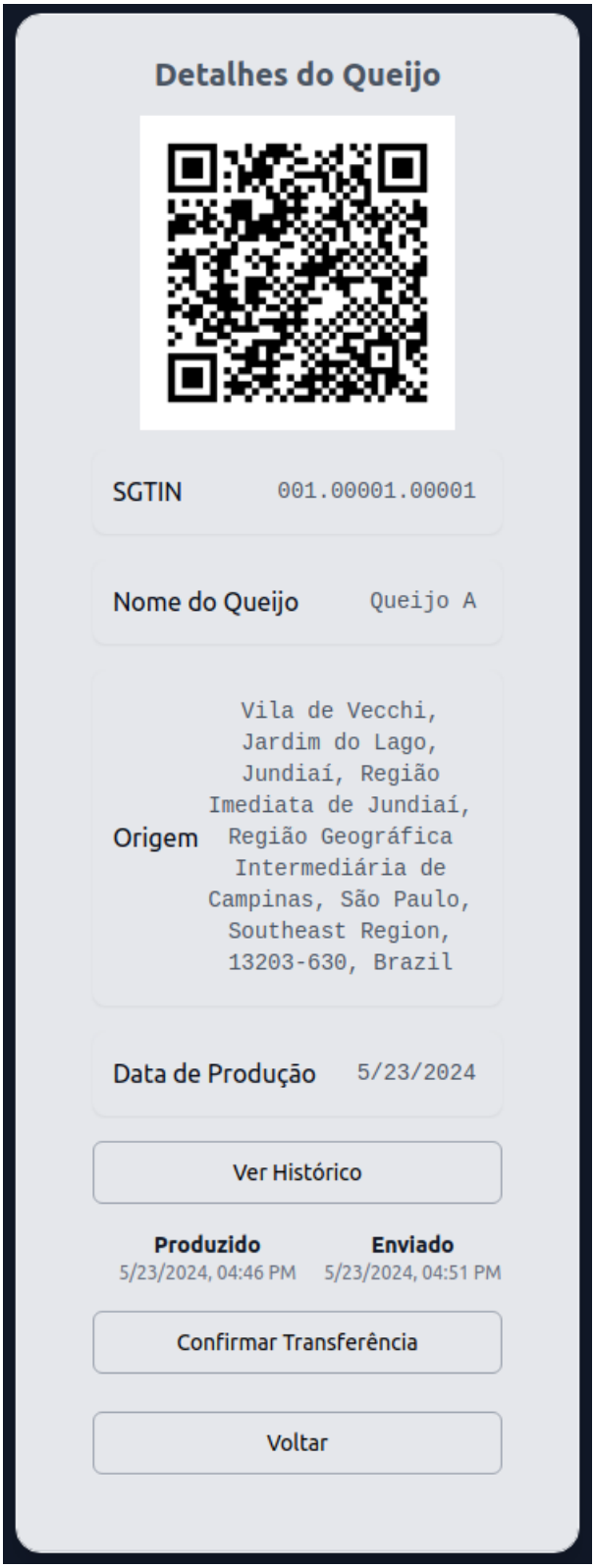

Figura 29 – Tela do queijo aguardando confirmação do varejista - Fonte: Autor

Assim que o varejista confirma o recebimento, o estado do queijo vai para "Entregue". Como pode ser visto na Figura 30, há uma nova opção para o varejista, o botão "Vender".

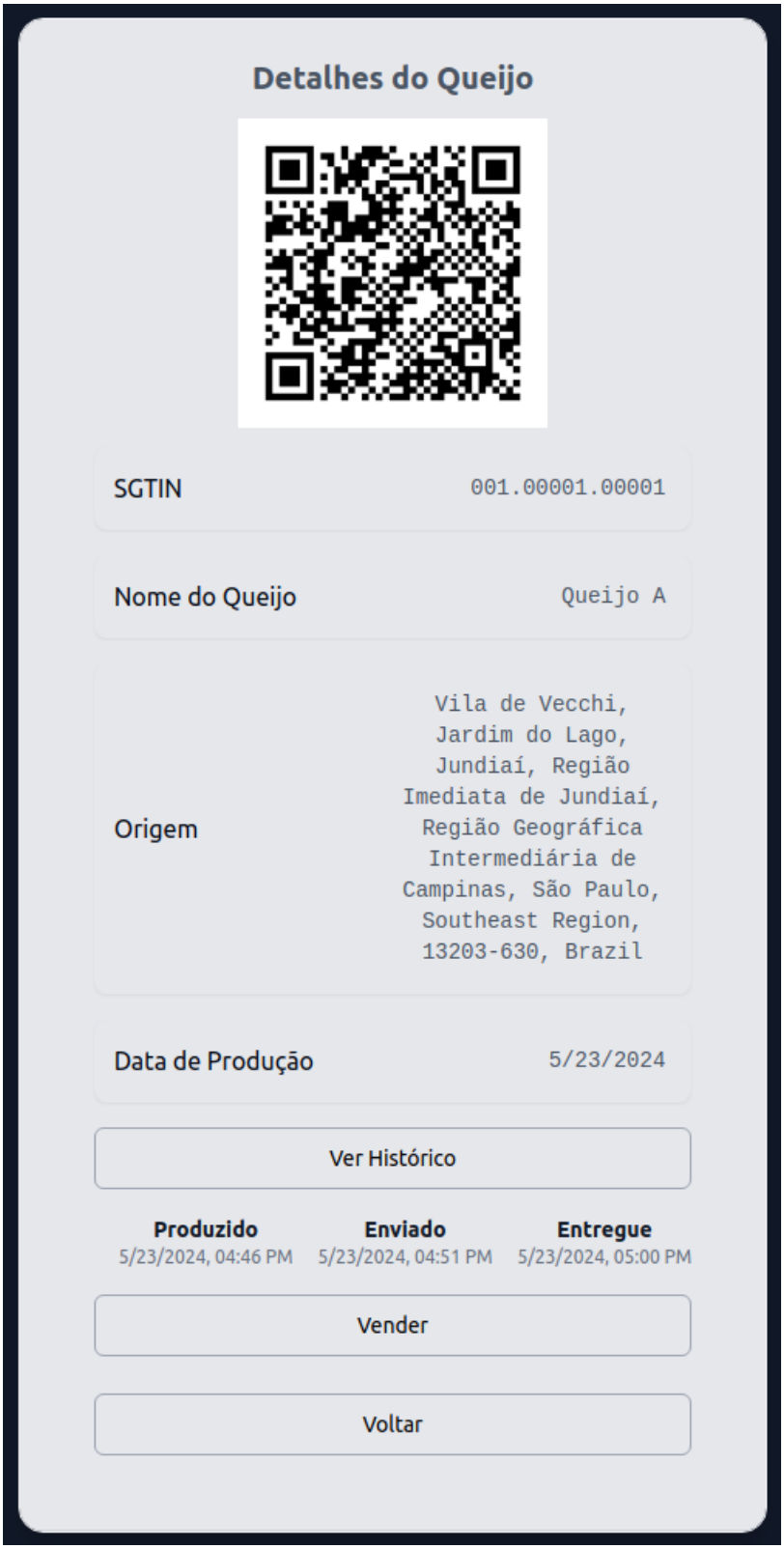

Figura 30 – Tela de detalhes do queijo após confirmado pelo varejista - Fonte: Autor

Após o varejista realizar a venda para o consumidor final, ele deve selecionar a opção "Vender", o que atualizará o queijo como vendido (Figura 31). A partir desse momento, não existem mais ações possíveis a serem realizadas por nenhum dos usuários.

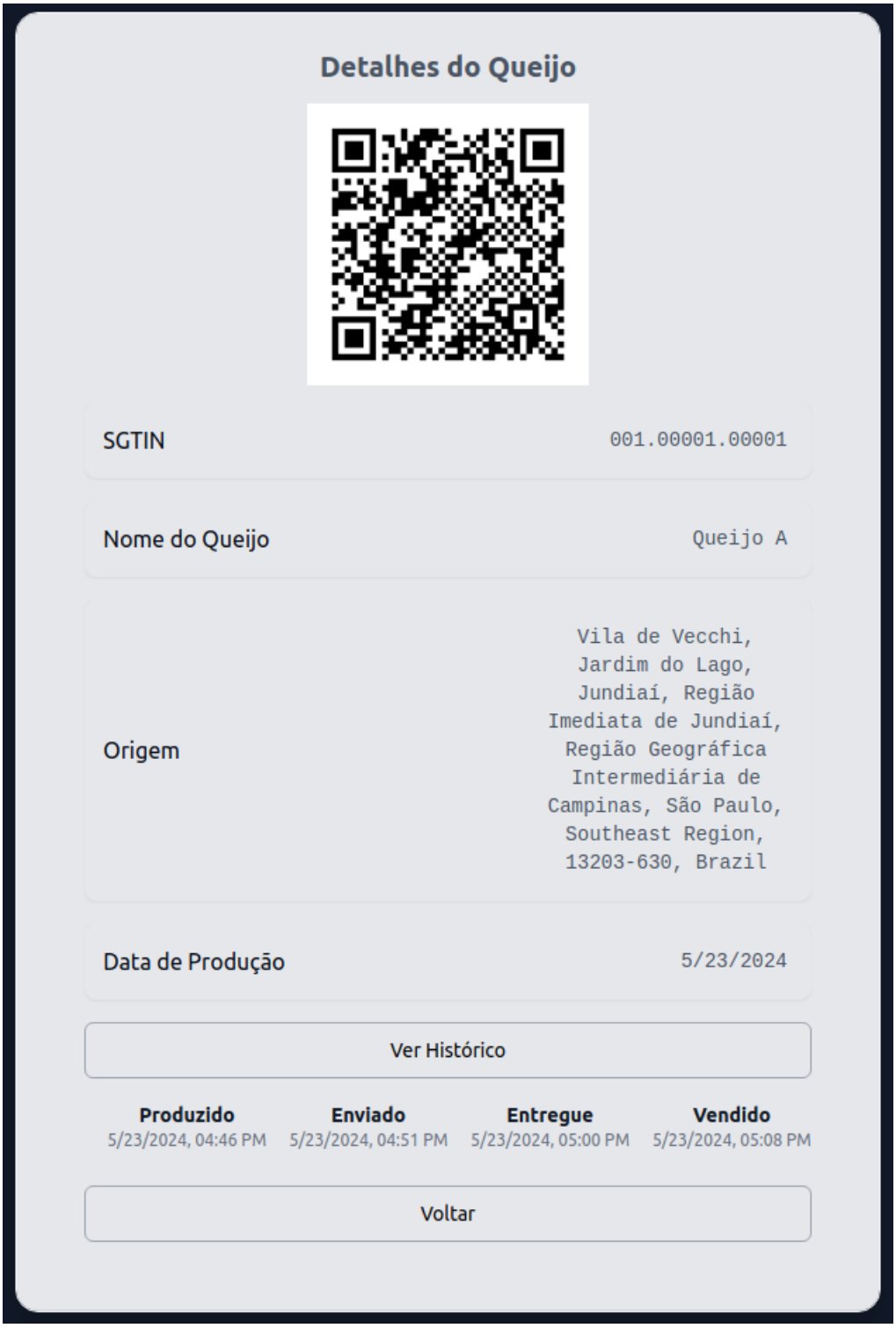

Figura 31 – Tela de detalhes do queijo após vendido - Fonte: Autor

O consumidor final, através da câmera do seu celular, poderá escanear o código QR da peça de queijo que comprou, tendo acesso à página mostrada na Figura 32 sem precisar de um cadastro na rede. Nesse caso, não há botão de "Voltar", pois não existe uma sessão ativa.

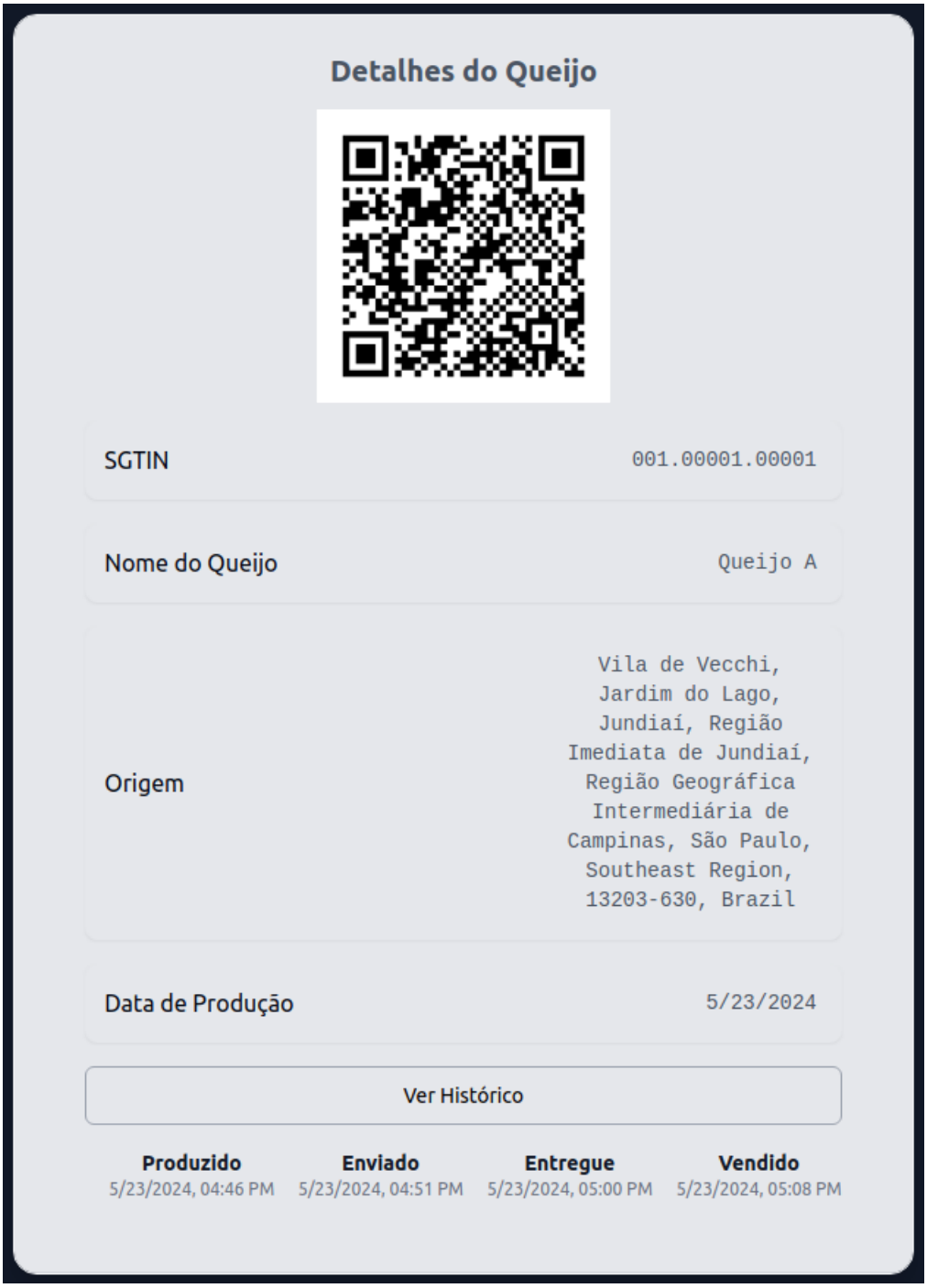

Figura 32 – Tela de queijo vista por um consumidor - Fonte: Autor

4.2.2 Back-end e API

O servidor *back-end* da nossa aplicação será responsável por receber as requisições do *front-end* e invocar as funções necessárias do *chaincode*, através de uma API, sendo desenvolvidos com o ambiente *runtime* Node.JS (OPENJS FOUNDATION, 2023). Nesse caso, foi decidido desenvolver uma API com arquitetura Representational State Transfer (REST), também conhecida como API RESTful, devido à sua simplicidade e escalabili-

dade. Essa arquitetura utiliza os métodos HTTP padrão (GET, POST, PUT e DELETE) e suporta diferentes formatos de dados, incluindo o que será amplamente utilizado nesse projeto, o formato JavaScript Object Notation (JSON).

Para criação da API, foi utilizada a ferramenta Express.js (OPENJS FOUNDATION, 2024a), um *framework* web para Node.js. Essa ferramenta foi escolhida por sua facilidade em definir rotas para requisições HTTP com suporte a middleware e por permitir a criação de APIs RESTful de forma ágil. Em conjunto com o Express, foi utilizado o cliente HTTP *Promise-based* Axios (AXIOS, 2024), que permite compatibilidade entre as requisições HTTP tanto no navegador quanto em Node.js. Além disso, essa biblioteca nos fornece uma API extremamente fácil de usar, especialmente pela transformação automática de dados JSON, que são amplamente utilizados nesse projeto.

Quando o usuário realiza uma ação no *front-end*, uma requisição é enviada com o método POST pelo Axios, sendo recebida pela API. A Figura 33 mostra algumas das funções do arquivo "index.ts", responsável por escutar as requisições e fazer as chamadas para o *back-end*.

```
36  app.post('/registerUser', async (req: Request, res: Response) => {
37    const { email, role } = req.body;
38    try {
39      const response = await registerUserOnLedger(email, role);
40      res.json(response);
41    } catch (error) {
42      res.status(500).json({ error });
43    }
44  });
45
46  app.post('/registerProduct', async (req: Request, res: Response) => {
47    const { email, productName } = req.body;
48    try {
49      const response = await registerProductOnLedger(email, productName);
50      res.json(response);
51    } catch (error) {
52      res.status(500).json({ error });
53    }
54  });
55
56  app.post('/getAllProducts', async (req: Request, res: Response) => {
57    const { email } = req.body;
58    try {
59      const response = await getAllUserProducts(email);
60      const products = JSON.parse(response);
61      res.json(products);
62    } catch (error) {
63      res.status(500).json({ error });
64    }
65  });
```

Figura 33 – Trecho de código da API - Fonte: Autor

Podemos observar na figura as chamadas de registro de usuário, registro de produto e consulta de todos os produtos de um determinado usuário. O corpo da requisição sempre traz o e-mail, para identificação do usuário que está executando a ação, e mais informações quando necessário. A partir disso, a respectiva função do *back-end* é chamada, e a resposta é devolvida para o *front-end*. As funções para registro de usuário e registro de produto podem ser vistas na Figura 34.

```
19  export async function registerUserOnLedger(
20    email: string,
21    role: Role
22  ): Promise<void> {
23    const ccp = await readCCP(role);
24    const walletPath = await buildWalletPath(role);
25    const wallet = await Wallets.newFileSystemWallet(walletPath);
26    const adminIdentity = await wallet.get('admin');
27    if (!adminIdentity) {
28      await enrollAdmin(role);
29    }
30    try {
31      const userExists = await wallet.get(email);
32      if (userExists) {
33        throw new Error(`User ${email} already exists in the wallet`);
34      }
35      await registerUser(wallet, ccp, email, role);
36    } catch (error) {
37      throw new Error(`Error registering user ${email}: ${error}`);
38    }
39  }
40
41  export async function registerProductOnLedger(
42    email: string,
43    productName: string
44  ): Promise<string> {
45    const gateway = await connectToGateway(email, 'supplier');
46    try {
47      const network = await gateway.getNetwork('my-channel1');
48      const contract = network.getContract(
49        'chaincode1',
50        'CheeseSupplyChainContract'
51      );
52      const result = await contract.submitTransaction(
53        'createProduct',
54        email,
55        productName
56      );
57      const productId = result.toString();
58      console.log(
59        `Product ${productName} successfully created with ID ${productId}`
60      );
61      gateway.disconnect();
62      return productId;
63    } catch (error) {
64      throw new Error(`Failed to submit transaction: ${error}`);
65    }
66  }
```

Figura 34 – Trecho de código do back-end - Fonte: Autor

Para o registro de um usuário no *ledger*, precisamos antes ler a configuração do perfil

daquele papel em específico, criar uma carteira e inscrever um administrador, caso ainda não exista. As funções para isso podem ser vistas na Figura 35.

```
13  export async function readCCP(role: Role): Promise<Record<string, any>> {
14    const __filename = fileURLToPath(import.meta.url);
15    const __dirname = path.dirname(__filename);
16    const ccpPath = path.resolve(__dirname, '..', '..', 'config', `${role}.json`);
17    const ccpJSON = fs.readFileSync(ccpPath, 'utf8');
18    return JSON.parse(ccpJSON);
19  }
20
21  export function buildWalletPath(role: Role): string {
22    return path.join(process.cwd(), `${role}-wallet`);
23  }
24
25  export async function connectToGateway(
26    email: string,
27    role: Role
28  ): Promise<Gateway> {
29    const ccp = await readCCP(role);
30    const walletPath = buildWalletPath(role);
31    const wallet = await Wallets.newFileSystemWallet(walletPath);
32    const identity = await wallet.get(email);
33    if (!identity) {
34      throw new Error(`User with email ${email} is not registered in the wallet`);
35    }
36    const gateway = new Gateway();
37    await gateway.connect(ccp, {
38      wallet,
39      identity: email,
40      discovery: { enabled: true, asLocalhost: true },
41    });
42    return gateway;
43  }
44
45  export async function enrollAdmin(role: Role) {
46    console.log('Enrolling admin...');
47    try {
48      const ccp = await readCCP(role);
49      const CAInfo = ccp.certificateAuthorities[`ca.${role}.supplychain.com`];
50      const TLSCACerts = CAInfo.tlsCACerts.pem;
51      const CA = new FabricCAServices(
52        CAInfo.url,
53        { trustedRoots: TLSCACerts, verify: false },
54        CAInfo.caName
55      );
56      const walletPath = await buildWalletPath(role);
57      const wallet = await Wallets.newFileSystemWallet(walletPath);
58      const isAdminEnrolled = await wallet.get('admin');
59      if (isAdminEnrolled) {
60        console.log('Admin already enrolled');
61        return;
62      }
63      const enrollment = await CA.enroll({
64        enrollmentID: 'admin',
65        enrollmentSecret: 'adminpw',
66      });
67      const identity: X509Identity = {
68        credentials: {
69          certificate: enrollment.certificate,
70          privateKey: enrollment.key.toBytes(),
71        },
72        type: 'X.509',
73        mspId: `${role}MSP`,
74      };
75      await wallet.put('admin', identity);
76      console.log('Successfully enrolled admin!');
77      if (role === 'supplier') {
78        await registerUser(wallet, ccp, 'dummy@mock.com', role);
79      }
```

Figura 35 – Trecho de código do back-end - Fonte: Autor

Como pode ser visto, para inscrever um administrador, precisamos das informações da CA, para então criar uma identidade que é um certificado digital X.509, com o MSP do papel exercido por esse administrador. Caso o papel seja de produtor, então haverá o registro de um usuário fictício, tratado aqui como *dummy*. Sua função é apenas a de conectar no *gateway* quando não existe uma sessão ativa, para que seja possível ver os detalhes de uma peça de queijo. Isso é necessário pois como já explicado, nossa rede *blockchain* é permissionada, sendo assim, toda chamada de função do *chaincode* deve ser feita por um usuário identificado na rede. A Figura 36 mostra a função de registro de usuário.

```typescript
export async function registerUser(
  wallet: Wallet,
  // eslint-disable-next-line @typescript-eslint/no-explicit-any
  ccp: Record<string, any>,
  email: string,
  role: Role
): Promise<string> {
  try {
    const CAInfo = ccp.certificateAuthorities[`ca.${role}.supplychain.com`];
    const TLSCACerts = CAInfo.tlsCACerts.pem;
    const CA = new FabricCAServices(
      CAInfo.url,
      { trustedRoots: TLSCACerts, verify: false },
      CAInfo.caName
    );
    const adminIdentity = await wallet.get('admin');
    if (!adminIdentity) {
      throw new Error(
        'An identity for the admin user "admin" does not exist in the wallet'
      );
    }
    const gateway = await connectToGateway('admin', role);
    const network = await gateway.getNetwork(channelName);
    const contract = network.getContract(chaincodeName, 'UserContract');
    const supplierIdBuffer = await contract.submitTransaction(
      'registerUser',
      email,
      role
    );
    const supplierId = supplierIdBuffer.toString();
    console.log(
      `Successfully registered ${role} with email ${email}${
        supplierId ? ' and ID ' + supplierId : ''
      }`
    );
    const provider = wallet
      .getProviderRegistry()
      .getProvider(adminIdentity.type);
    const adminUser = await provider.getUserContext(adminIdentity, 'admin');
    const secret = await CA.register(
      {
        affiliation: `${role}.department1`,
        enrollmentID: email,
        role: 'client',
      },
      adminUser
    );
    const enrollment = await CA.enroll({
      enrollmentID: email,
      enrollmentSecret: secret,
    });
    const userIdentity: X509Identity = {
      credentials: {
        certificate: enrollment.certificate,
        privateKey: enrollment.key.toBytes(),
      },
      mspId: `${role}MSP`,
      type: 'X.509',
    };
    await wallet.put(email, userIdentity);
    console.log(`Successfully imported ${role} ${email} into the wallet`);
    return supplierId;
  } catch (error) {
    throw new Error(`Failed to register ${role} ${email}: ${error}`);
  }
}
```

Figura 36 – Trecho de código para registro de usuário - Fonte: Autor

Esse registro é feito de maneira similar ao do administrador, porém, no caso do administrador ele é inscrito diretamente pela CA. Depois que esse administrador é registrado na rede, ele pode se conectar ao *gateway* para registrar novos usuários. Note que uma identidade X.509 também é gerada para os usuários que são registrados na carteira. Quando as funções mostradas na Figura 30 são executadas com sucesso, recebemos uma mensagem no console indicando isso, como pode ser visto na Figura 37.

```
0|index    | Enrolling admin...
0|index    | Successfully enrolled admin!
0|index    | Successfully registered supplier with email dummy@mock.com and ID 000000000
0|index    | Successfully imported supplier dummy@mock.com into the wallet
0|index    | Successfully registered supplier with email jose@test.com and ID 000000001
0|index    | Successfully imported supplier jose@test.com into the wallet
0|index    | Product Queijo A successfully created with ID 00000000100000000001
0|index    | Cheese lot successfully created with ID 0000000010000000000100000000000001
```

Figura 37 – Logs de registro de produtor e produto - Fonte: Autor

Nesse caso, como o usuário "jose@test.net" foi o primeiro produtor a se cadastrar na rede, primeiramente houve a inscrição de um administrador, e também do produtor fictício. Ele recebe um ID de produtor **000000000**, pois não deve participar ativamente da rede. Então, o usuário é registrado com ID **000000001**, e a partir daí registra o produto Queijo A com GTIN **00000000100000000001** e um lote de queijo, a partir desse produto, com ID **0000000010000000000100000000000001**.

Para o banco de dados que guarda as informações de cadastro do usuário, foi feita a escolha de um banco de dados relacional. Bancos de dados relacionais são ideais para esse caso em que existe uma estrutura definida, visto que os dados de usuário seguem um esquema específico (IBM, 2022). O banco de dados escolhido foi o PostgreSQL (POSTGRESQL, 2024), por ser um banco de dados relacional que simplifica a organização dos dados.

O Object Relational Mapper (ORM) escolhido para criação da tabela necessária foi o DrizzleORM (DRIZZLEORM, 2024) que, apesar de relativamente novo, possui poderosas ferramentas para gerenciar os objetos de maneira leve e rápida. A Figura 38 mostra o esquema criado para esse banco, com informações como nome, e-mail e senha.

```typescript
import { pgTable, serial, text, varchar } from 'drizzle-orm/pg-core';

export const users = pgTable('User', {
  id: serial('id').primaryKey(),
  name: text('name').notNull(),
  email: varchar('email', { length: 64 }),
  password: varchar('password', { length: 64 }),
  role: text('role').notNull(),
  cnpj: text('cnpj').notNull(),
});
```

Figura 38 – Esquema do banco de dados relacional - Fonte: Autor

4.2.3 Rede Blockchain

Para a rede *blockchain* da nossa aplicação, por motivos já mencionados anteriormente, escolhemos a utilização da ferramenta Hyperledger Fabric, na versão 2.5.6. Para referências futuras, importantes mudanças foram implementadas em sua última versão beta (v3). Visando manter o padrão, a linguagem utilizada para desenvolvimento dos contratos inteligentes também foi TypeScript.

Segundo a documentação de Hyperledger Fabric (HYPERLEDGER FOUNDATION, 2024a), o primeiro passo para iniciar um desenvolvimento em Hyperledger Fabric é a criação de uma rede. Essa é uma etapa complexa, pois requer a criação de um arquivo *docker-compose*, arquivos YAML e *scripts*. Quando qualquer mudança é feita no design, como adição de um novo *peer* ou organização, todos esses arquivos precisam ser manualmente e cuidadosamente atualizados. Para fins de criação do MVP, neste projeto foi utilizada a ferramenta Fablo (SOFTWAREMILL, 2024), que facilita a criação de uma rede a partir de apenas um arquivo JSON chamado *fablo-config.json*. Nesse arquivo, são definidas as informações como organizações, *peers*, canais, *chaincode* e controle de versões. A partir desse arquivo, Fablo cria uma pasta *fablo-target* com todos os arquivos necessários para funcionamento da rede, tornando o processo muito mais simples: basta alterar esse único arquivo quando quisermos atualizar nossa rede, e as mudanças serão refletidas assim que iniciarmos a rede novamente.

Com isso em mente, foi criada uma rede simples, com três organizações: produtor, distribuidor e varejista, cada uma delas contendo um único *peer*. O *chaincode* conta com dois contratos inteligentes, sendo um para as funções de usuário, mostradas na Figura 39, e outro para as funções dos queijos. O banco de dados escolhido para o *chaincode* foi o CouchDB, que possui fácil integração com Hyperledger Fabric e possui suporte a *rich queries* em JSON, permitindo consultas complexas com base em campos específicos (APACHE, 2024).

```typescript
import { Context, Contract } from 'fabric-contract-api';

interface User {
  email: string;
  role: string;
  supplierId?: string;
}

const supplierIdBits = 9;

export class UserContract extends Contract {
  public async registerUser(
    ctx: Context,
    email: string,
    role: string
  ): Promise<string | undefined> {
    let user: User = { email, role };
    if (role === 'supplier') {
      user.supplierId = await this.generateSupplierId(ctx);
    }
    await ctx.stub.putState(email, Buffer.from(JSON.stringify(user)));
    console.log(
      `Successfully created user with email ${email}${
        user.supplierId ? ' and ID ' + user.supplierId : ''
      }`
    );
    return user.supplierId;
  }

  public async generateSupplierId(ctx: Context): Promise<string> {
    const supplierIdKey = 'supplierIdCounter';
    const counterBytes = await ctx.stub.getState(supplierIdKey);
    let counter =
      counterBytes && counterBytes.length > 0
        ? parseInt(counterBytes.toString())
        : -1;
    counter++;
    await ctx.stub.putState(supplierIdKey, Buffer.from(counter.toString()));
    const binaryString = counter.toString(2).padStart(supplierIdBits, '0');
    return binaryString;
  }

  public async getUser(ctx: Context, email: string): Promise<string> {
    const user = await ctx.stub.getState(email);
    if (!user || user.length === 0) {
      throw new Error(`The user with e-mail ${email} does not exist`);
    }
    const userJSON = JSON.parse(user.toString());
    return userJSON;
  }
}
```

Figura 39 – Código do contrato inteligente para funções do usuário - Fonte: Autor

Como podemos observar, foi definida uma interface de usuário com seu e-mail, papel e um ID de produtor opcional, pois apenas produtores terão esse ID. Caso seja feito o registro de um usuário cujo papel seja de produtor, é chamada a função para gerar seu ID,

de maneira sequencial, a partir de um contador iniciado em 0 (para o produtor fictício). Além disso, existe uma função para pegar um usuário baseado em seu e-mail.

Já no contrato inteligente para funções do queijo, temos na Figura 40 o exemplo de uma função, a que registra um produto.

```
62  const productIdBits = 11;
63  const serialIdBits = 76;
64  const cheeseLotSerialIdBits = 15;
65
66  export class CheeseSupplyChainContract extends Contract {
67    public async createProduct(
68      ctx: Context,
69      email: string,
70      productName: string
71    ): Promise<string> {
72      const userAsBytes = await ctx.stub.getState(email);
73      if (!userAsBytes || userAsBytes.length === 0) {
74        throw new Error(`User with email ${email} does not exist`);
75      }
76      const user = JSON.parse(userAsBytes.toString());
77      if (user.role !== 'supplier' || !user.supplierId) {
78        throw new Error(
79          `Only Suppliers are allowed to create products. User is ${user.role}`
80        );
81      }
82      const supplierId = user.supplierId;
83      const productCounterKey = `productCounter_${user.supplierId}`;
84      const productCounterBytes = await ctx.stub.getState(productCounterKey);
85      const productCounter =
86        productCounterBytes && productCounterBytes.length > 0
87          ? parseInt(productCounterBytes.toString()) + 1
88          : 1;
89      await ctx.stub.putState(
90        productCounterKey,
91        Buffer.from(productCounter.toString())
92      );
93      const productId = productCounter.toString(2).padStart(productIdBits, '0');
94      const GTIN = `${supplierId}${productId}`;
95      const product: Product = {
96        type: 'product',
97        supplierId,
98        productId,
99        GTIN,
100       productName,
101     };
102     await ctx.stub.putState(GTIN, Buffer.from(JSON.stringify(product)));
103     const labels = await this.createLabels(
104       ctx,
105       supplierId,
106       productId,
107       productName
108     );
109     console.log(
110       `Sucessfully registered product ${productName} with ID ${GTIN} and generated ${labels.length} labels`
111     );
112     return GTIN;
113   }
```

Figura 40 – Código do contrato inteligente para criação de produto - Fonte: Autor

Primeiro, definimos quantos bits serão usados para o produto, código serial e ID serial do lote. Então, a função tenta buscar um usuário na carteira a partir de seu e-mail

e verifica se ele é um produtor. Caso seja, é gerada uma chave para guardar o ID do produto, que começa em 1. Tendo disponível o ID do produtor e do produto, podemos então gerar nosso GTIN como a junção de ambos, e criar nosso produto, que é salvo no banco de dados. Além disso, há a chamada da função que cria etiquetas, para que estas sejam geradas assim que o produto é registrado. Por fim, a função deve registrar no console que o produto foi registrado com sucesso, bem como a geração de etiquetas, o que pode ser visto na Figura 41.

```
func2 -> Sucessfully registered product Queijo A with ID 0000000010000000001 and generated 50 labels
```

Figura 41 – Registro do docker com mensagem de sucesso - Fonte: Autor

Capítulo 5

Homologação

Após a criação do MVP, foram realizados testes para garantir seu pleno funcionamento e medir seu desempenho. Todos os testes abaixo foram realizados localmente, em uma máquina virtual com acesso a 12GB de memória RAM e 6 *threads* de um processador Ryzen 7 3700X, no sistema operacional Linux Mint.

5.1 Testes de Funcionamento

Para garantir que a parte funcional do sistema está de acordo com os requisitos, foram realizados testes de integração, utilizando um *framework* para testes em Node.js chamado Mocha, que permite a execução assíncrona e sequencial dos testes de maneira simples (OPENJS FOUNDATION, 2024b), juntamente com a biblioteca de asserções Chai, que fornece interfaces e tipos para facilitar a verificação dos resultados esperados nos testes (CHAIJS, 2024). Com essa combinação, foi possível criar, estruturar e validar os testes do back-end, a partir de testes das chamadas de API para as funções do *chaincode*.

```
import chaiHttp from 'chai-http';
import { use, expect } from 'chai';
import { describe, it } from 'mocha';

const chai = use(chaiHttp);
const serverUrl = 'http://localhost:4000';

const random = Math.floor(Math.random() * 1000);
const supplierTestEmail = `supplier${random}@test.com`;
const distributorTestEmail = `distributor${random}@test.com`;
const retailerTestEmail = `retailer${random}@test.com`;
let GTIN: string;
let cheeseLotId: string;
let SGTIN: string;

describe('Cheese Supply Chain API Success Cases', () => {
  it('should register a new supplier', async () => {
    const user = {
      email: supplierTestEmail,
      role: 'supplier',
    };
    await chai
      .request(serverUrl)
      .post('/registerUser')
      .send(user)
      .then(res => {
        expect(res).to.have.status(200);
      })
      .catch();
  });
```

Figura 42 – Importação de bibliotecas e registro de produtor - Fonte: Autor

Como mostra a Figura 42, basta importar as bibliotecas Mocha e Chai e apontar a URL em que o *back-end* está rodando para começar os testes. Primeiramente, definimos um número aleatório entre 1 e 1000, que é utilizado para gerar e-mails diferentes a cada execução. Isso é importante pois estamos mandando requisições para o servidor e registrando usuários no *ledger*, logo, devemos evitar que o mesmo usuário seja registrado mais de uma vez, pois essa tentativa deve falhar. Definimos o ID do lote, o GTIN e o SGTIN fora do escopo de testes para que possamos reutilizar seus valores nos outros testes. Por fim, temos nosso primeiro teste, o de registro de produtor, em que mandamos um usuário para a função de registro e esperamos receber o *status* 200 (OK).

```
100
101  it('should register a new cheese', async () => {
102    SGTIN = `${GTIN}00000000000000000000000000000000000000000000000000000000000000000000000000000001`;
103    const cheese = {
104      SGTIN,
105      email: supplierTestEmail,
106      cheeseLotId: cheeseLotId,
107    };
108    return chai
109      .request(serverUrl)
110      .post('/registerCheese')
111      .send(cheese)
112      .then(res => {
113        expect(res).to.have.status(200);
114        const cheese = JSON.parse(res.body.response);
115        expect(cheese.SGTIN).to.equal(SGTIN);
116        expect(cheese.cheeseLotId).to.equal(cheeseLotId);
117        expect(cheese.currentState).to.equal('PRODUCED');
118        expect(cheese.ownerId).to.contain(supplierTestEmail);
119      })
120      .catch();
121  });
122
123  it('should transfer a cheese from supplier to distributor', async () => {
124    const req = {
125      SGTIN,
126      email: supplierTestEmail,
127      role: 'supplier',
128      newOwnerEmail: distributorTestEmail,
129      newOwnerRole: 'distributor',
130    };
131    return chai
132      .request(serverUrl)
133      .post('/transferCheese')
134      .send(req)
135      .then(res => {
136        expect(res).to.have.status(200);
137        const cheese = JSON.parse(res.body);
138        expect(cheese.SGTIN).to.equal(SGTIN);
139        expect(cheese.cheeseLotId).to.equal(cheeseLotId);
140        expect(cheese.currentState).to.equal('PENDING');
141        expect(cheese.ownerId).to.contain(supplierTestEmail);
142        expect(cheese.pendingOwnerId).to.contain(distributorTestEmail);
143      })
144      .catch();
145  });
```

Figura 43 – Registro de queijo e transferência para o distribuidor - Fonte: Autor

Na Figura 43, no teste do registro de queijo, temos verificações para garantir que ele está sendo criado com sucesso. Por exemplo, o SGTIN e o ID do lote devem bater com o que mandamos para a função, o *status* deve ser "Produzido", e o campo de ID do dono deve conter o e-mail do produtor. No teste de transferência para o distribuidor, verificamos as mesmas informações, porém o *status* deve ser "Pendente", e o ID do dono pendente deve conter o e-mail do distribuidor.

```
147  it('should confirm receipt of cheese by distributor', async () => {
148    const req = {
149      SGTIN,
150      email: distributorTestEmail,
151      role: 'distributor',
152    };
153    return chai
154      .request(serverUrl)
155      .post('/confirmCheeseTransfer')
156      .send(req)
157      .then(res => {
158        expect(res).to.have.status(200);
159        const cheese = JSON.parse(res.body);
160        expect(cheese.SGTIN).to.equal(SGTIN);
161        expect(cheese.cheeseLotId).to.equal(cheeseLotId);
162        expect(cheese.currentState).to.equal('SHIPPED');
163        expect(cheese.ownerId).to.contain(distributorTestEmail);
164      })
165      .catch();
166  });
```

Figura 44 – Confirmação de transferência pelo distribuidor - Fonte: Autor

A Figura 44 mostra o teste feito para confirmação de transferência do queijo pelo distribuidor. Nesse caso, o estado atual do queijo deve ser "Enviado", enquanto o ID do dono precisa conter o e-mail do distribuidor, visto que a posse do queijo deve ter passado para ele.

Serão testados também casos de falha, para garantir que ações que não devem ser permitidas estejam resultando em erro. Na Figura 45, podemos ver dois desses casos: no primeiro, tentamos registrar um usuário que já está registrado, utilizando o mesmo e-mail de produtor. Já no segundo caso, testamos o registro de usuário com uma função que não existe, ou seja, diferente de produtor, distribuidor e varejista. Nesses casos, esperamos o retorno do *status* 500.

```
254  describe('Cheese Supply Chain API Failure Cases', () => {
255    it('should fail to register existing user', async () => {
256      const user = {
257        email: supplierTestEmail,
258        role: 'supplier',
259      };
260      return chai
261        .request(serverUrl)
262        .post('/registerUser')
263        .send(user)
264        .then(res => {
265          expect(res).to.have.status(500);
266        })
267        .catch();
268    });
269
270    it('should fail to register a new user due to invalid role', async () => {
271      const user = {
272        email: 'random@email',
273        role: 'undefined',
274      };
275      return chai
276        .request(serverUrl)
277        .post('/registerUser')
278        .send(user)
279        .then(res => {
280          expect(res).to.have.status(500);
281        })
282        .catch();
283    });
```

Figura 45 – Casos de falha de registro de usuário - Fonte: Autor

Para garantir que o projeto esteja funcionando corretamente, rodaremos o teste esperando uma cobertura de 100%. Além dos resultados dos testes, veremos também o tempo de resposta entre a chamada de API e o fim da execução das funções, dessa forma poderemos analisar se o tempo está dentro do aceitável para uso.

5.2 Testes de Desempenho

Os testes de performance têm como objetivo verificar o desempenho do sistema, simulando vários usuários submetendo transações na rede ao mesmo tempo. Para isso, foi utilizado o Hyperledger Caliper. Caliper é uma ferramenta de *benchmarking* de performance para algumas plataformas *blockchain*, sendo compatível com Hyperledger Fabric (HYPERLEDGER FOUNDATION, 2024b). O funcionamento da ferramenta se dá a partir de dois arquivos de configuração YAML, um de rede e um de *benchmark*, e arquivos de *workload* escritos em JavaScript, contendo as instruções para requisição das transações no *chaincode*. A Figura 46 mostra parte do arquivo *network.yaml* com as especificações da rede:

```yaml
name: Caliper
version: '2.0.0'

caliper:
  blockchain: fabric

channels:
  - channelName: my-channel1
    contracts:
      - id: chaincode1
    orderers:
      - orderer0.group1.orderer.supplychain.com

peers:
  peer0.supplier.supplychain.com:
    url: grpc://localhost:7041
  peer0.distributor.supplychain.com:
    url: grpc://localhost:7061
  peer0.retailer.supplychain.com:
    url: grpc://localhost:7081

organizations:
  - mspid: supplierMSP
    identities:
      certificates:
        - name: 'User1'
          clientPrivateKey:
            path: '../fablo-target/fabric-config/crypto-config/peerOrganizations/supplier.supplych
          clientSignedCert:
            path: '../fablo-target/fabric-config/crypto-config/peerOrganizations/supplier.supplych
    connectionProfile:
      path: '../fablo-target/fabric-config/connection-profiles/connection-profile-supplier.yaml'
      discover: true
  - mspid: distributorMSP
    identities:
      certificates:
        - name: 'User1'
          clientPrivateKey:
            path: '../fablo-target/fabric-config/crypto-config/peerOrganizations/distributor.suppl
```

Figura 46 – Parte do arquivo network.yaml - Fonte: Autor

Como pode-se observar, é necessário especificar a plataforma em que serão feitos os testes, as informações dos canais, os *peers* com seus respectivos endereços e as organizações, com o nome de um usuário para cada e sua respectiva CA e chave privada para autenticação. Na Figura 47 é mostrada parte do arquivo *benchmark.yaml*.

```yaml
test:
  name: cheese-supply-chain-benchmark
  description: Benchmark for Cheese Supply Chain
  workers:
    type: local
    number: 5
  rounds:
    - label: register-suppliers
      description: Register suppliers
      txNumber: 100
      rateControl:
        type: fixed-rate
        opts:
          tps: 10
      workload:
        module: workloads/registerSuppliers.js

    - label: register-distributors
      description: Register distributors
      txNumber: 100
      rateControl:
        type: fixed-rate
        opts:
          tps: 10
      workload:
        module: workloads/registerDistributors.js

    - label: register-retailers
      description: Register retailers
      txNumber: 100
      rateControl:
        type: fixed-rate
        opts:
          tps: 10
      workload:
        module: workloads/registerRetailers.js

    - label: create-products
      description: Create cheese products
      txNumber: 100
```

Figura 47 – Parte do arquivo benchmark.yaml - Fonte: Autor

O arquivo deve conter o número de trabalhadores (*workers*) que irão submeter as transações, cada um deles utilizando uma *thread* do processador. Além disso, para cada rodada de testes (chamadas aqui de *rounds*), devemos definir quantas transações serão feitas e em qual taxa, e apontar o arquivo de *workload* que contém as instruções das requisições.

Com esses dois arquivos criados, pode-se criar os arquivos de workload para efetivamente testar a rede. Já no primeiro teste, registro de usuários, foi encontrado um problema que se perpetuou nos outros testes, o erro *MVCC_READ_CONFLICT*. Esse erro de MultiVersion Concurrency Control (MVCC) ocorre quando existe um conflito de leitura e escrita, normalmente causado por duas ou mais transações que tentam ler e modificar o mesmo dado simultaneamente, gerando um problema de concorrência: caso um dado tenha sido modificado entre a leitura inicial e a tentativa de escrita, a transação será rejeitada.

Como o Hyperledger Caliper paraleliza as transações usando múltiplos *workers*, todos eles vão ler o estado atual dos dados e tentar escrever a modificação. No entanto, apenas o primeiro consegue, pois os outros verão que o estado foi alterado pelo primeiro *worker*, sendo assim, a versão dos dados não corresponde mais a versão atual, gerando o erro. No nosso *chaincode*, esse erro ocorria em todas as transações que incluíam modificar uma chave sequencial (como os IDs dos produtores, por exemplo).

A solução encontrada foi alterar as funções do *chaincode* com algumas adaptações, dentre elas, receber o ID como parâmetro. Assim, cada *worker* chama a função com o valor do ID já definido, resolvendo o problema. Por exemplo, no caso do ID dos produtores, a função de registro será chamada com um valor de ID igual ao número da transação atual; assim, caso sejam feitas 100 transações, serão gerados IDs de 1 a 100, sem chance de conflito. Essas adaptações foram feitas exclusivamente para teste de carga da rede, pois em uma aplicação real, os IDs devem ser gerados dentro do *chaincode*, por motivos de segurança.

A Figura 48 mostra o arquivo *registerSuppliers.js*, responsável pelo registro dos produtores na rede. Esse arquivo foi criado tendo como base os próprios exemplos fornecidos pela Hyperledger em seu repositório *caliper-benchmarks* (HYPERLEDGER, 2024), tendo um funcionamento simples. Primeiro, definimos uma classe para registro dos produtores, iniciando o índice da transação em 0. Cada vez que uma transação é submetida, através da função *submitTransaction()*, incrementamos esse índice, que é usado como ID do produtor. Geramos, também, um e-mail único para cada produtor com base nesse índice. Então, a função de registro de usuários é executada com os argumentos de e-mail, função e ID do produtor.

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

const role = 'supplier';

class RegisterSuppliersWorkload extends WorkloadModuleBase {
  constructor() {
    super();
    this.txIndex = 0;
  }

  async submitTransaction() {
    this.txIndex++;
    const supplierId = this.txIndex;
    const email = `supplier${this.txIndex}@example.com`;
    const args = {
      contractId: 'chaincode1',
      contractFunction: 'UserContract:registerUser',
      contractArguments: [email, role, supplierId],
    };
    await this.sutAdapter.sendRequests(args);
  }
}

function createWorkloadModule() {
  return new RegisterSuppliersWorkload();
}

module.exports.createWorkloadModule = createWorkloadModule;
```

Figura 48 – Arquivo registerSuppliers.js - Fonte: Autor

As funções de registro de distribuidor e varejista seguem a mesma lógica, porém sem a necessidade de um ID de produtor. Os outros arquivos seguem uma lógica similar, como pode ser visto na Figura 49, que mostra o arquivo *createProducts.js*. Nele, os índices das transações também são utilizados para definir qual produtor criará o produto. Como nesse caso teremos cada produtor criando seu primeiro produto, passamos o ID do produto como sendo 1.

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

const role = 'supplier';

class RegisterSuppliersWorkload extends WorkloadModuleBase {
  constructor() {
    super();
    this.txIndex = 0;
  }

  async submitTransaction() {
    this.txIndex++;
    const supplierId = this.txIndex;
    const email = `supplier${this.txIndex}@example.com`;
    const args = {
      contractId: 'chaincode1',
      contractFunction: 'UserContract:registerUser',
      contractArguments: [email, role, supplierId],
    };
    await this.sutAdapter.sendRequests(args);
  }
}

function createWorkloadModule() {
  return new RegisterSuppliersWorkload();
}

module.exports.createWorkloadModule = createWorkloadModule;
```

Figura 49 – Arquivo createProducts.js - Fonte: Autor

A mesma lógica segue para os outros arquivos de teste. Com os arquivos criados e a rede em funcionamento, podemos utilizar o seguinte comando para iniciar os testes a partir da pasta em que o Caliper está configurado: **npx caliper launch manager –caliper-workspace . –caliper-benchconfig benchmark.yaml –caliper-networkconfig network.yaml**. Esse comando executa os testes da maneira definida em nosso arquivo *benchmark.yaml* e retorna um arquivo *report.html*, contendo as informações de latência dos testes.

Como forma de medir o desempenho da rede, será utilizada como base a monografia "Blockchain, LAI e LGPD: Uma Proposta de Uso Responsável de Blockchain no Contexto de Acesso a Dados do Cidadão e Garantia do Direito à Transparência" (MENDES, 2022), na seção 3.5.1, em que foram realizados testes similares, também utilizando Hyperledger Caliper.

No nosso caso, também utilizaremos 5 *workers*, mas começaremos os testes com uma

taxa de 5 transações por segundo (TPS), para cada um dos 11 *rounds* (um por arquivo). Os próximos testes serão feitos dobrando essa taxa, sempre com um limite de 100 transações, até que a rede comece a apresentar sinais de saturação. Cada um dos testes mencionados será executado três vezes, e a média desses testes serão apresentados no próximo capítulo.

As funções de criação e consulta de peças de queijo serão, presumidamente, as mais utilizadas na rede. Para estas, serão feitos testes com diferentes valores de transações por segundo (10, 50, 100, 200 e 500), tendo como objetivo simular uma possível carga na rede em horários de mais uso. O resultado desse teste será representado por um gráfico, mostrado na seção de Resultados.

Capítulo 6

Resultados

6.1 Teste de Funcionamento

A Figura 50 mostra o resultado da execução do arquivo de testes, feito com Mocha e a biblioteca Chai.

```
Cheese Supply Chain API Success Cases
  ✓ should register a new supplier (3686ms)
  ✓ should register a new distributor (3681ms)
  ✓ should register a new retailer (3530ms)
  ✓ should register a new product (3450ms)
  ✓ should register a new cheese lot (3000ms)
  ✓ should register a new cheese (3908ms)
  ✓ should transfer a cheese from supplier to distributor (2886ms)
  ✓ should confirm receipt of cheese by distributor (3002ms)
  ✓ should transfer a cheese from distributor to retailer (2838ms)
  ✓ should confirm receipt of cheese by retailer (3286ms)
  ✓ should sell a cheese (3162ms)
  ✓ should get the cheese information (382ms)

Cheese Supply Chain API Failure Cases
  ✓ should fail to register existing user
  ✓ should fail to register a new user due to invalid role
  ✓ should fail to register a new cheese lot due to invalid role (43ms)
  ✓ should fail to transfer a cheese from retailer to supplier (202ms)


16 passing (37s)
```

Figura 50 – Tela de impressão contendo duas etiquetas - Fonte: Autor

Todos os casos de teste passaram com sucesso, indicando que o funcionamento do sistema está de acordo com o esperado. Na frente de cada teste, podemos ver seu tempo de resposta individualmente. As funções que fazem alteração no *ledger*, em geral, levam cerca de 3 a 4 segundos entre o momento em que a API é chamada até sua execução e recebimento da resposta, enquanto o tempo de consulta de um queijo foi de apenas 382 milissegundos.

Esses resultados são satisfatórios, visto que as funções não apresentam um grande atraso para o usuário final.

6.2 Teste de Desempenho

A Tabela 1 mostra as médias dos valores obtidos a partir de três testes realizados com 5 transações por segundo (TPS). Os dados individuais de cada teste pode ser visto no Anexo 1.

Função	**Latência Max. (s)**	**Latência Mín. (s)**	**Latência Média (s)**	**Throughput (TPS)**
register-suppliers	2.37	0.64	1.47	5.0
register-distributors	2.16	0.67	1.42	5.0
register-retailers	2.24	0.58	1.47	4.93
create-products	14.28	2.00	6.83	3.13
create-cheese-lots	2.18	0.58	1.37	5.0
create-cheeses	2.23	0.68	1.42	4.97
transfer-cheeses-to-distributors	2.29	0.65	1.49	4.97
confirm-transfers-to-distributors	2.32	0.67	1.45	5.0
transfer-cheeses-to-retailers	2.20	0.65	1.42	5.0
confirm-transfers-to-retailers	2.18	0.61	1.32	5.07
sell-cheeses	2.14	0.64	1.38	5.03
get-cheeses	1.80	0.40	1.12	5.1

Tabela 1 – Médias dos testes realizados com 5 TPS - Fonte: Autor

Como pode-se observar, as latências das funções estão por volta de 1,5 segundos e o *throughput* desejado de 5 TPS foi obtido com sucesso, com exceção da função de criação de produtos.

Função	**Latência Max. (s)**	**Latência Mín. (s)**	**Latência Média (s)**	**Throughput (TPS)**
register-suppliers	1.67	0.60	1.14	9.47
register-distributors	2.09	0.63	1.27	9.17
register-retailers	1.91	0.70	1.28	9.40
create-products	28.41	2.10	12.86	2.87
create-cheese-lots	1.97	0.65	1.17	9.40
create-cheeses	1.79	0.69	1.23	9.33
transfer-cheeses-to-distributors	2.12	0.64	1.27	9.27
confirm-transfers-to-distributors	1.86	0.71	1.26	9.27
transfer-cheeses-to-retailers	1.72	0.67	1.19	9.43
confirm-transfers-to-retailers	1.74	0.57	1.14	9.37
sell-cheeses	2.12	0.73	1.34	9.37
get-cheeses	1.32	0.40	0.86	9.8

Tabela 2 – Médias dos testes realizados com 10 TPS - Fonte: Autor

Observando a Tabela 2, podemos inferir que as funções continuam com latência similar quando submetemos 10 transações por segundo.

Função	**Latência Max. (s)**	**Latência Mín. (s)**	**Latência Média (s)**	**Throughput (TPS)**
register-suppliers	3.47	0.73	2.03	12.87
register-distributors	3.33	0.93	2.42	12.20
register-retailers	4.61	0.82	2.61	11.47
create-products	41.21	2.69	20.78	2.23
create-cheese-lots	2.82	0.74	1.77	13.93
create-cheeses	3.80	0.82	2.23	12.40
transfer-cheeses-to-distributors	3.69	0.68	2.10	12.47
confirm-transfers-to-distributors	3.53	0.81	2.06	12.80
transfer-cheeses-to-retailers	3.02	0.77	1.91	13.57
confirm-transfers-to-retailers	3.23	0.86	1.93	13.87
sell-cheeses	3.96	0.85	2.44	12.10
get-cheeses	1.59	0.38	0.90	17.2

Tabela 3 – Médias dos testes realizados com 20 TPS - Fonte: Autor

No teste em que submetemos 20 transações por segundo, visto na Tabela 3, podemos verificar que os valores de latência média aumentam em relação aos outros testes, indicando saturação da rede *blockchain*. Apesar disso, eles continuam aceitáveis, visto que não chegam aos 3 segundos, novamente desconsiderando a função "create-products".

O alto tempo de resposta da função que cria produtos se deve à geração das 50 etiquetas após o produto ser criado, que são escritas individualmente no *ledger*, para que o usuário possa imprimi-las. Uma possível solução seria criar etiquetas uma a uma, gerando uma nova sempre que a anterior fosse usada, mas isso não permitiria que o usuário imprimisse mais de uma etiqueta por vez. Considerando que um produtor dificilmente irá utilizar

essa função muitas vezes, esse tempo não deve impactar o fluxo do sistema de forma significativa.

As funções com maior probabilidade de serem chamadas muitas vezes são, como mencionado anteriormente, a de criação de peças de queijo (a partir de um lote) e a de consulta de uma peça de queijo. Para esses casos, obtivemos os seguintes valores de latência média:

Função	**10 TPS (s)**	**50 TPS (s)**	**100 TPS (s)**	**200 TPS (s)**	**500 TPS (s)**
create-cheeses	1.23	9.51	16.03	16.12	17.63
get-cheeses	0.86	2.01	6.34	6.94	7.40

Tabela 4 – Médias das latências com diferentes TPS - Fonte: Autor

O gráfico mostrado na Figura 51 ilustra os diferentes valores de latência, em segundos, para os diferentes valores de transações por segundo.

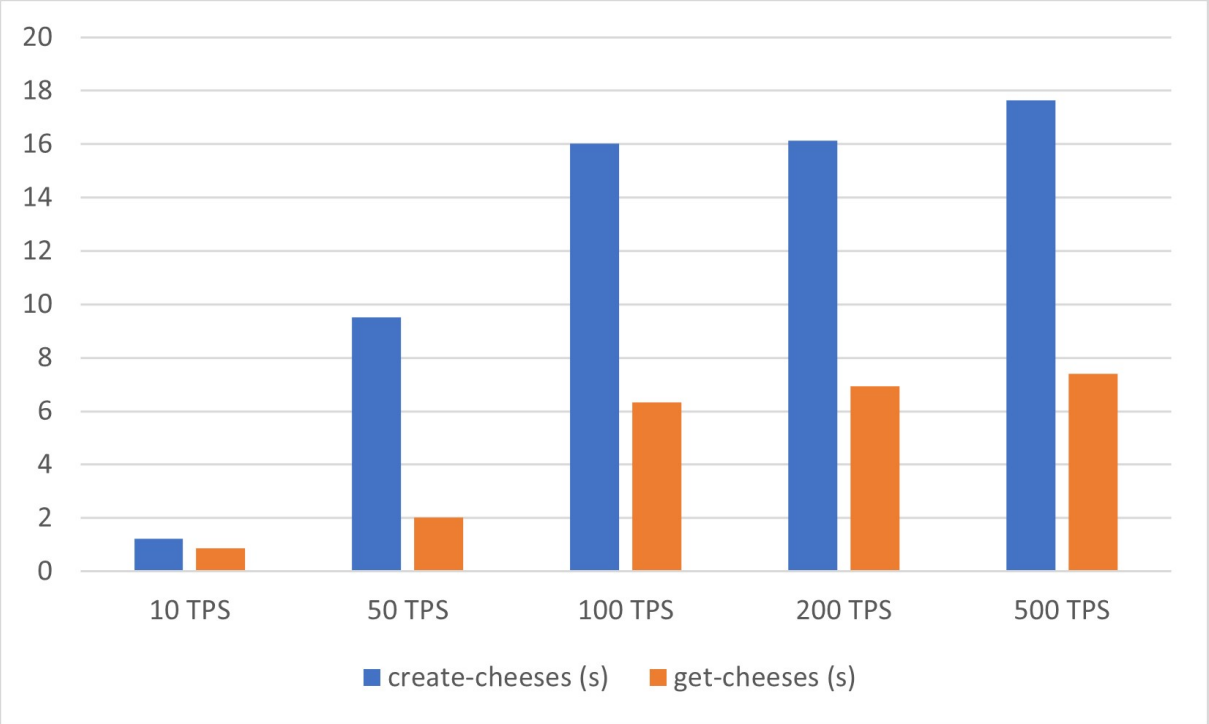

Figura 51 – Gráfico com valores das latências para diferentes TPS - Fonte: Autor

Podemos observar que quando submetemos 10 transações por segundo, os tempos são baixos, indicando que a rede não tem problemas em lidar com essa carga. Já em 50 TPS, temos um aumento significativo, principalmente na função de criação das peças de queijo. A partir de 100 TPS, temos outro aumento significativo no tempo de resposta, dessa vez também na função de consulta das peças, que tem seu tempo triplicado em relação a 50 TPS.

Capítulo 7

Conclusão

Observando os requisitos do projeto e os resultados dos testes, podemos concluir que os objetivos foram cumpridos com sucesso na criação do nosso MVP. O sistema criado possui todas as características necessárias para ser um produto mínimo viável, visto que permite todo o fluxo do processo, desde o cadastro dos usuários até a venda das peças de queijo, e a tecnologia *blockchain* se mostrou muito útil em cadeias de produção como essa, pois aumenta a transparência na rastreabilidade de todo o processo.

Tudo isso foi feito com tecnologias modernas e atuais, que possuem grandes comunidades ativas e são suportados por renomadas organizações, como Linux Foundation e Vercel, garantindo uma fácil manutenção no futuro, por meio de atualizações contínuas e documentação abrangente. Hyperledger Fabric permite suporte para criação de contratos inteligentes em linguagens modernas, como TypeScript, facilitando o desenvolvimento de *software*, enquanto Next.js facilita a integração e manutenção do *back-end* e *front-end* a partir da criação simplificada de APIs. O uso de TypeScript no projeto inteiro promove a reutilização de código e também facilita sua manutenção, visto que desenvolvedores não precisam aprender novas linguagens para continuar o desenvolvimento. Por fim, ferramentas como Mocha e Caliper para os testes garantem que o código possa ser continuamente integrado e testado, o que contribui para a manutenção do projeto.

O sistema roda sem grandes complicações, como esperado, tanto em computadores quanto em celulares, o que é um requisito essencial do projeto. Para garantir o funcionamento correto do sistema, os testes de integração com Mocha foram essenciais para verificar a interação entre os diferentes componentes, incluindo casos de falha para uma cobertura mais ampla. Além disso, os testes de carga foram fundamentais para validar o desempenho da rede *blockchain* sob alta demanda, simulando múltiplos usuários simultâneos. Apesar da rede sofrer saturação quando exposta à muita carga, isso não traz um

impacto tão significativo ao usuário final, visto que os tempos de resposta permanecem dentro do aceitável nas funções que deverão ser utilizadas com mais frequência.

7.1 Trabalhos Futuros

Futuros trabalhos podem usar este como ponto de partida para expansão da tecnologia que foi desenvolvida aqui. Como já citado, esse projeto pode ser modificado sem grandes problemas para rastreabilidade de outros produtos que não sejam queijos artesanais, bastando fazer adaptações em algumas partes relevantes do código, como as organizações e o *chaincode*.

Como o foco deste trabalho foi a criação de um MVP, ou seja, um produto mínimo viável, existe bastante espaço para melhoria da solução desenvolvida nele. Como prioridade, a segurança dos dados dos usuários deve ser fortalecida por meio de criptografia caso essa solução venha a ser utilizada por usuários reais. Para fins de criação do MVP, por mais que tenham sido seguidas boas práticas de segurança, como o uso correto da biblioteca "next-auth" e autenticação dos usuários no *chaincode*, esse não foi o foco da aplicação, visto que não foram utilizados dados de pessoas reais durante os testes.

Na parte de código, existem também algumas possíveis melhorias a serem feitas. Em primeiro momento, seria interessante atualizar as tecnologias utilizadas para sua versão mais nova. Por exemplo, Hyperledger Fabric já possui uma versão 3 em fase beta, que traz mudanças na maneira de se conectar com a rede e invocar *chaincodes*, logo, uma continuação desse trabalho poderia envolver essa atualização.

A parte de usuário também poderia ser melhor trabalhada em atualizações futuras desse projeto, desde o registro até o uso. No registro, por exemplo, qualquer pessoa pode preencher as informações da tela e se cadastrar normalmente. Pensando em um cenário real, seria importante analisar os registros de usuário para saber que se tratam de produtores, distribuidores e varejistas reais, impedindo o acesso de pessoas indesejadas ao sistema. Em relação ao uso, adicionar possibilidades como edição de informações, inserção de foto de perfil e troca de e-mail e senha podem tornar a experiência do usuário mais agradável. Esses pontos não foram trabalhados neste projeto pois não eram tão importantes para criação do MVP.

Em relação a cadeia produtiva, este trabalho teve como foco um conjunto de *stakeholders* relacionados à cadeia de queijo. Em uma versão futura, a solução pode ser expandida para incluir processos anteriores na cadeia, adicionando novas informações de leite, animais, pasto e outras partes do processo. Além disso, outra possível melhoria seria aumentar a quantidade de informações sobre um determinado lote ou peça de queijo. Por exemplo, integrar o rastreamento do transporte em tempo real pelo distribuidor a essa aplicação traria mais transparência do processo.

Por fim, uma versão prática da continuidade deste trabalho focaria nos seguintes passos:

1. Produzir uma versão para homologação em campo, que envolve revisão de código e disponibilização da plataforma para novos usuários;
2. Disponibilizar a plataforma para uso em período de testes para produtores, distribuidores, varejistas e consumidores de queijos artesanais;
3. Após um período de três a seis meses, homologar a solução para ser de fato utilizada em campo.

Referências

APACHE. **CouchDB**. 2024. Disponível em: <https://docs.couchdb.org/en/stable/>.

ASIF, R.; HASSAN, S. R. Shaping the future of ethereum: exploring energy consumption in proof-of-work and proof-of-stake consensus. **Frontiers in Blockchain**, v. 6, 2023. ISSN 2624-7852. Disponível em: <https://www.frontiersin.org/journals/blockchain/articles/10.3389/fbloc.2023.1151724>.

AXIOS. **Promise based HTTP client for the browser and node.js**. 2024. Disponível em: <https://axios-http.com/>.

AZZI, R.; CHAMOUN, R. K.; SOKHN, M. The power of a blockchain-based supply chain. **Computers Industrial Engineering**, v. 135, p. 582–592, 2019. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S0360835219303729>.

BACH, L. M.; MIHALJEVIC, B.; ZAGAR, M. Comparative analysis of blockchain consensus algorithms. **2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**, p. 1545–1550, 2018.

BUTERIN, V. On public and private blockchains. **Ethereum Foundation Blog**, 2015. Disponível em: <https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains>.

CHAIJS. **Chai**. 2024. Disponível em: <https://www.chaijs.com/>.

DELOITTE. **Global Powers of Retailing 2023**. 2023. Disponível em: <https://www.deloitte.com/global/en/Industries/consumer/analysis/global-powers-of-retailing.html>. Acesso em: 23.11.2023.

DRIZZLEORM. **DrizzleORM**. 2024. Disponível em: <https://orm.drizzle.team/>.

Freepik. **Cheese, Delivery, Store, Female, Computer icons**. 2024. <https://www.flaticon.com/authors/freepik>. Accessed: 2024-08-02.

HUANG, D.; MA, X.; ZHANG, S. Performance analysis of the raft consensus algorithm for private blockchains. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, v. 50, n. 1, 2020.

HYPERLEDGER. **Hyperledger Caliper Benchmarks**. 2024. Disponível em: <https://github.com/hyperledger/caliper-benchmarks>.

HYPERLEDGER FOUNDATION. **A Blockchain Platform for the Enterprise**. 2024. Disponível em: <https://hyperledger-fabric.readthedocs.io/en/latest/>.

_____. **Caliper**. 2024. Disponível em: <https://www.hyperledger.org/projects/caliper>.

IBM. **SQL vs. NoSQL Databases: What's the Difference?** 2022. Disponível em: <https://www.ibm.com/blog/sql-vs-nosql/>.

LASHKARI, B.; MUSILEK, P. A comprehensive review of blockchain consensus mechanisms. **IEEE Access**, v. 9, p. 43620–43652, 2021.

LIN, Q. et al. Food safety traceability system based on blockchain and epcis. **IEEE Access**, v. 7, p. 20698–20707, 2020. Disponível em: <https://ieeexplore.ieee.org/document/8640818>.

LINUX FOUNDATION. **Hyperledger - The Open Global Ecosystem for Enterprise Blockchain**. 2024. Disponível em: <https://www.hyperledger.org/>.

MENDES, L. P. **Blockchain, LAI e LGPD: Uma Proposta de Uso Responsável de Blockchain no Contexto de Acesso a Dados do Cidadão e Garantia do Direito à Transparência**. Dissertação (Trabalho de Conclusão de Curso) — Universidade Federal de São Carlos, São Carlos, SP, 2022.

MICROSOFT. **TypeScript: JavaScript With Syntax For Types**. 2023. Disponível em: <https://www.typescriptlang.org/>.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. 2009. Disponível em: <https://bitcoin.org/bitcoin.pdf>.

NEXT-PWA. **next-pwa**. 2024. Disponível em: <https://github.com/shadowwalker/next-pwa>.

NEXTAUTH.JS. **NextAuth.js**. 2024. Disponível em: <https://next-auth.js.org/>.

NEXT.JS. **Next.js**. 2024. Disponível em: <https://nextjs.org/>.

OLSEN, P.; BORIT, M. How to define traceability. **Trends in Food Science Technology**, v. 29, n. 2, p. 142–150, 2013. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S0924224412002117>.

OPENJS FOUNDATION. **Node.js**. 2023. Disponível em: <https://nodejs.org/en>.

_____. **Express.js**. 2024. Disponível em: <https://expressjs.com/>.

_____. **Mocha - the fun, simple, flexible JavaScript test framework**. 2024. Disponível em: <https://mochajs.org/>.

PIERRO, M. D. What is the blockchain? **Computing in Science Engineering**, v. 19, n. 5, p. 92–95, 2017.

POSTGRESQL. **PostgreSQL**. 2024. Disponível em: <https://www.postgresql.org/>.

SCHOLTEN, H. et al. Defining and analyzing traceability systems in food supply chains. In: ESPIÑEIRA, M.; SANTACLARA, F. J. (Ed.). **Advances in Food Traceability Techniques and Technologies**. Woodhead Publishing, 2016, (Woodhead Publishing Series in Food Science, Technology and Nutrition). p. 9–33. ISBN 978-0-08-100310-7. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780081003107000028>.

SEBRAE. **Relatório de Inteligência: Queijos Artesanais**. Santa Catarina, 2021. Disponível em: <https://www.sebrae-sc.com.br/observatorio/relatorio-de-inteligencia/comercializacao-de-queijos-artesanais>. Acesso em: 24.10.2023.

SOFTWAREMILL. **fablo**. 2024. Disponível em: <https://github.com/hyperledger-labs/fablo>.

SOON, T. J. Qr code. **synthesis journal**, v. 2008, p. 59–78, 2008.

STEEN, M. V.; TANENBAUM, A. S. **Distributed Systems**. 4th. ed. [S.l.]: distributed-systems.net, 2023.

TRIPATHI, G.; AHAD, M. A.; CASALINO, G. A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges. **Decision Analytics Journal**, v. 9, p. 100344, 2023. ISSN 2772-6622. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2772662223001844>.

UDDIN, M. Blockchain medledger: Hyperledger fabric enabled drug traceability system for counterfeit drugs in pharmaceutical industry. **International Journal of Pharmaceutics**, v. 597, n. 120235, 2021. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0378517321000399>.

VALENTE, F.; PAULINO, L. The right of transparency for citizens in the context of gdpr using blockchain: a proposed architecture. **Proceedings of SpliTech 2024**, 2024.

VEVLE, G. et al. **Guideline for Unique identification of products with SGTIN (serialized GTIN). Labelling with GS1 Datamatrix barcode and tagging with EPC / RFID Gen 2 UHF RFID tags**. Norway, 2018.

WALMART. **How Walmart brought unprecedented transparency to the food supply chain with Hyperledger Fabric**. 2019. Disponível em: <https://www.hyperledger.org/case-studies/walmart-case-study>.

Anexos

ANEXO A

Testes do Hyperledger Caliper

A.1 Testes com 5 TPS

Name	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
register-suppliers	2.31	0.66	1.49	5.0
register-distributors	2.10	0.56	1.33	5.0
register-retailers	2.25	0.53	1.45	4.9
create-products	14.18	2.10	6.16	3.1
create-cheese-lots	2.09	0.52	1.30	5.0
create-cheeses	2.25	0.71	1.42	5.0
transfer-cheeses-to-distributors	2.00	0.58	1.30	5.1
confirm-transfers-to-distributors	2.00	0.51	1.29	5.0
transfer-cheeses-to-retailers	2.25	0.67	1.43	5.0
confirm-transfers-to-retailers	2.04	0.55	1.26	5.1
sell-cheeses	2.10	0.57	1.37	5.1
get-cheeses	1.82	0.41	1.13	5.1

Tabela 5 – Teste 1 de 3 com 5 transações por segundo (TPS) - Fonte: Autor

Name	**Max Latency (s)**	**Min Latency (s)**	**Avg Latency (s)**	**Throughput (TPS)**
register-suppliers	2.56	0.56	1.46	5.0
register-distributors	2.22	0.65	1.45	5.0
register-retailers	2.31	0.69	1.51	4.9
create-products	15.07	1.75	7.17	3.1
create-cheese-lots	2.27	0.69	1.42	5.0
create-cheeses	2.12	0.68	1.44	5.0
transfer-cheeses-to-distributors	2.64	0.73	1.72	4.9
confirm-transfers-to-distributors	2.64	0.84	1.64	5.0
transfer-cheeses-to-retailers	2.16	0.65	1.40	5.0
confirm-transfers-to-retailers	2.29	0.61	1.30	5.1
sell-cheeses	2.15	0.70	1.38	5.0
get-cheeses	1.76	0.45	1.11	5.1

Tabela 6 – Teste 2 de 3 com 5 transações por segundo (TPS) - Fonte: Autor

Name	**Max Latency (s)**	**Min Latency (s)**	**Avg Latency (s)**	**Throughput (TPS)**
register-suppliers	2.23	0.71	1.46	5.0
register-distributors	2.16	0.80	1.47	5.0
register-retailers	2.15	0.52	1.44	5.0
create-products	13.58	2.15	7.16	3.2
create-cheese-lots	2.18	0.54	1.39	5.0
create-cheeses	2.33	0.64	1.41	4.9
transfer-cheeses-to-distributors	2.23	0.65	1.44	4.9
confirm-transfers-to-distributors	2.33	0.65	1.43	5.0
transfer-cheeses-to-retailers	2.18	0.64	1.44	5.0
confirm-transfers-to-retailers	2.21	0.68	1.39	5.0
sell-cheeses	2.17	0.64	1.40	5.0
get-cheeses	1.81	0.34	1.13	5.1

Tabela 7 – Teste 3 de 3 com 5 transações por segundo (TPS) - Fonte: Autor

A.2 Testes com 10 TPS

Name	**Max Latency (s)**	**Min Latency (s)**	**Avg Latency (s)**	**Throughput (TPS)**
register-suppliers	1.51	0.62	1.08	9.5
register-distributors	2.85	0.73	1.40	8.7
register-retailers	1.89	0.69	1.31	9.4
create-products	25.81	2.02	12.46	2.9
create-cheese-lots	2.83	0.78	1.47	8.9
create-cheeses	2.11	0.86	1.43	9.0
transfer-cheeses-to-distributors	2.30	0.76	1.32	9.3
confirm-transfers-to-distributors	1.76	0.83	1.28	9.4
transfer-cheeses-to-retailers	1.82	0.81	1.32	9.2
confirm-transfers-to-retailers	1.80	0.65	1.18	9.3
sell-cheeses	2.90	0.82	1.59	9.2
get-cheeses	1.28	0.37	0.84	9.7

Tabela 8 – Teste 1 de 3 com 10 transações por segundo (TPS) - Fonte: Autor

Name	**Max Latency (s)**	**Min Latency (s)**	**Avg Latency (s)**	**Throughput (TPS)**
register-suppliers	1.91	0.63	1.23	9.2
register-distributors	1.64	0.56	1.15	9.5
register-retailers	1.70	0.74	1.20	9.5
create-products	20.86	2.12	9.30	3.5
create-cheese-lots	1.56	0.60	1.05	9.5
create-cheeses	1.66	0.66	1.17	9.4
transfer-cheeses-to-distributors	1.75	0.69	1.18	9.3
confirm-transfers-to-distributors	2.01	0.56	1.27	9.1
transfer-cheeses-to-retailers	1.61	0.64	1.10	9.7
confirm-transfers-to-retailers	1.63	0.54	1.03	9.4
sell-cheeses	1.78	0.64	1.25	9.4
get-cheeses	1.28	0.38	0.83	9.8

Tabela 9 – Teste 2 de 3 com 10 transações por segundo (TPS) - Fonte: Autor

Name	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
register-suppliers	1.58	0.55	1.12	9.7
register-distributors	1.78	0.61	1.26	9.3
register-retailers	2.15	0.68	1.32	9.3
create-products	38.56	2.15	16.83	2.2
create-cheese-lots	1.51	0.58	1.00	9.8
create-cheeses	1.60	0.56	1.10	9.6
transfer-cheeses-to-distributors	2.31	0.48	1.30	9.2
confirm-transfers-to-distributors	1.82	0.73	1.24	9.3
transfer-cheeses-to-retailers	1.72	0.58	1.16	9.4
confirm-transfers-to-retailers	1.80	0.53	1.21	9.4
sell-cheeses	1.68	0.74	1.19	9.5
get-cheeses	1.39	0.44	0.91	9.8

Tabela 10 – Teste 3 de 3 com 10 transações por segundo (TPS) - Fonte: Autor

A.3 Testes com 20 TPS

Name	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
register-suppliers	3.88	0.78	2.29	12.3
register-distributors	3.86	1.01	2.48	12.2
register-retailers	3.52	0.80	2.30	12.9
create-products	40.50	2.66	20.57	2.3
create-cheese-lots	2.70	0.73	1.67	13.8
create-cheeses	4.08	0.80	2.35	11.9
transfer-cheeses-to-distributors	3.05	0.68	1.89	13.2
confirm-transfers-to-distributors	3.49	0.86	2.10	12.8
transfer-cheeses-to-retailers	3.22	0.77	2.07	13.1
confirm-transfers-to-retailers	2.64	0.94	1.66	14.5
sell-cheeses	4.33	0.73	2.48	11.4
get-cheeses	1.58	0.26	0.98	16.4

Tabela 11 – Teste 1 de 3 com 20 transações por segundo (TPS) - Fonte: Autor

Name	**Max Latency (s)**	**Min Latency (s)**	**Avg Latency (s)**	**Throughput (TPS)**
register-suppliers	3.14	0.77	1.91	13.6
register-distributors	5.03	0.87	2.82	10.8
register-retailers	4.05	0.67	2.41	11.8
create-products	47.72	3.74	25.07	1.9
create-cheese-lots	3.13	0.94	2.05	13.7
create-cheeses	3.77	0.81	2.14	12.5
transfer-cheeses-to-distributors	4.54	0.70	2.18	11.5
confirm-transfers-to-distributors	3.47	0.86	2.03	12.9
transfer-cheeses-to-retailers	3.66	0.68	2.12	12.5
confirm-transfers-to-retailers	4.30	0.96	2.47	11.3
sell-cheeses	3.97	0.93	2.42	12.3
get-cheeses	1.40	0.37	0.81	18.0

Tabela 12 – Teste 2 de 3 com 20 transações por segundo (TPS) - Fonte: Autor

Name	**Max Latency (s)**	**Min Latency (s)**	**Avg Latency (s)**	**Throughput (TPS)**
register-suppliers	3.40	0.65	1.89	12.7
register-distributors	3.11	0.90	1.95	13.6
register-retailers	6.26	0.98	3.14	9.7
create-products	35.41	1.67	16.71	2.5
create-cheese-lots	2.62	0.55	1.59	14.3
create-cheeses	3.54	0.85	2.21	12.8
transfer-cheeses-to-distributors	3.47	0.67	2.22	12.7
confirm-transfers-to-distributors	3.63	0.69	2.04	12.7
transfer-cheeses-to-retailers	2.17	0.85	1.53	15.1
confirm-transfers-to-retailers	2.76	0.68	1.65	13.8
sell-cheeses	3.58	0.88	2.43	12.6
get-cheeses	1.78	0.51	0.91	17.3

Tabela 13 – Teste 3 de 3 com 20 transações por segundo (TPS) - Fonte: Autor