

UNIVERSITY OF THE PHILIPPINES MANILA
COLLEGE OF ARTS AND SCIENCES
DEPARTMENT OF PHYSICAL SCIENCES AND MATHEMATICS

A BLOCKCHAIN-BASED SYSTEM FOR SUPPLY CHAIN
GOVERNANCE IN THE PHILIPPINE PIG INDUSTRY

A special problem in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science

Submitted by:

Kyle Mari Angelo M. Aquino

June 2023

Permission is given for the following people to have access to this SP:

Available to the general public	Yes
Available only after consultation with author/SP adviser	No
Available only to those bound by confidentiality agreement	No

ACCEPTANCE SHEET

The Special Problem entitled “A blockchain-based system for supply chain governance in the Philippine pig industry” prepared and submitted by Kyle Mari Angelo M. Aquino in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science has been examined and is recommended for acceptance.

Marbert John C. Marasigan, M.Sc. (*cand.*)
Adviser

EXAMINERS:

	Approved	Disapproved
1. Avegail D. Carpio, M.Sc.	_____	_____
2. Richard Bryann L. Chua, Ph.D. (<i>cand.</i>)	_____	_____
3. Perlita E. Gasmen, M.Sc. (<i>cand.</i>)	_____	_____
4. Ma. Sheila A. Magboo, Ph.D. (<i>cand.</i>)	_____	_____
5. Vincent Peter C. Magboo, M.D., M.Sc.	_____	_____
6. Geoffrey A. Solano, Ph.D.	_____	_____

Accepted and approved as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Vio Jianu C. Mojica, M.Sc.
Unit Head
Mathematical and Computing Sciences Unit
Department of Physical Sciences
and Mathematics

Marie Josephine M. De Luna, Ph.D.
Chair
Department of Physical Sciences
and Mathematics

Maria Constancia O. Carrillo, Ph.D.
Dean
College of Arts and Sciences

Abstract

An approximate 63% of total pork produce in the Philippines is sourced from small backyard operations. However, the lack of regulatory oversight and poor handling of biosecurity in these backyard farms lead to significant challenges in disease control and quality assurance. To address these issues, there is a need to establish a traceability-to-farm system that would encourage small backyard operators to participate in quality control processes. This system can also provide transparency to consumers and stakeholders alike and help establish trust and confidence in the supply chain. To this end, this paper has developed a blockchain-based solution for supply chain governance that leverages Hyperledger Fabric's enterprise-grade design to enable efficient and reliable tracking of pigs and pork products in a customizable and scalable manner.

Keywords: Blockchain, Supply Chain, Governance, Multisignature, Transparency, Traceability, Pure Utility

Contents

Acceptance Sheet	i
Abstract	ii
List of Figures	vi
List of Tables	viii
I. Introduction	1
A. Background of the Study	1
B. Statement of the Problem	2
C. Objectives of the Study	3
C..1 Blockchain System	3
C..2 Client Application	4
D. Significance of the Project	5
E. Scope and Limitations	6
F. Assumptions	7
II. Review of Related Literature	8
A. Supply Chain Governance	8
B. Overview of Blockchain Technology	9
C. Blockchain Technology as Applied to Supply Chains	11
C..1 Synthesis	13
III. Theoretical Framework	15
A. Overview of Pork Supply Chains in the Philippines	15
B. Blockchain Technology	15
C. Hyperledger Fabric	16
D. Apache CouchDB	18

E.	Azure	18
F.	Docker	19
G.	Kubernetes	19
H.	Hyperledger Caliper	19
IV.	Design and Implementation	21
A.	Blockchain Pork Supply Chain	21
B.	Use Case Design	23
B..1	Requirement Analysis	23
B..2	Design and Development	25
C.	Use Cases	31
D.	Database Design	32
E.	System Architecture	43
F.	Technical Architecture	45
V.	Results	47
A.	System Architecture Overview	47
A..1	Blockchain Network	47
A..2	Client Mobile Application	50
A..3	Main Server	52
A..4	Secondary Server	54
A..5	Off-chain Slave Database	55
B.	Client Mobile App Functionalities	55
B..1	Authentication	56
B..2	Adding a New Pig to the User's Location	58
B..3	Adding a New Product to the User's Location	59
B..4	Updating a Pig	60
B..5	Viewing Pig History	61

B..6	Viewing Product History	62
B..7	Adding a Pig Auction	63
B..8	Bidding in a Pig Auction	64
B..9	Accepting/Rejecting a Bid	65
B..10	Confirming a Pig Transfer	66
B..11	Confirming a Product Transfer	68
B..12	Registering a New User	69
B..13	Updating an Existing User	71
B..14	Removing a User from a Location	72
C.	Public API Service	73
VI.	Discussion	77
VII.	Conclusion	82
VIII.	Recommendations	84
IX.	Bibliography	87
X.	Appendix	92
A.	Smart Contracts	92
XI.	Acknowledgment	110

List of Figures

1	Blockchain-based supply chain governance structure with a multisig-nature protocol [1]	12
2	Relationship Dynamics between Value Chain Actor Respondents [2] .	15
3	Pork supply chain in Western Leyte [3]	22
4	PoA-based Multisignature pork supply chain architecture	27
5	The blockchain network	28
6	Use Case Diagrams for the PorkWatch client mobile application . . .	31
7	Activity Diagram for Logging in to the Mobile App	32
8	Activity Diagram for the Server-Blockchain Transaction Process . . .	32
9	Data Model	34
10	Docker containers for managing the kubernetes cluster and related images	47
11	K9s terminal based UI displaying the pods for the blockchain network	47
12	Transactions in the Blockchain (with custom console.log prints every invocation)	49
13	PorkWatch Mobile App Screens	50
14	Events Emitted by the Smart Contracts in the Blockchain Network .	54
15	Logging in	56
16	Adding a new pig	58
17	Adding a new product	59
18	Updating a pig	60
19	Viewing Pig History	61
20	Viewing Product History	62
21	Adding a Pig Auction	63
22	Bidding in a Pig Auction	64
23	Accepting a Bid	65
24	Confirming a Pig Transfer	66

25	Confirming a Product Transfer	68
26	Registering a New User pt.1	69
27	Registering a New User pt.2	70
28	Updating an Existing User	71
29	Removing a User from a Location	72
30	Dockerized Off-chain CouchDB	73
31	The Slave Database	73
32	Requesting all Products in the System	74
33	Requesting the Product with Id: 1 in Detail	75
34	Constructing a Stringified CouchDB Query Encoded in URI using JS- Fiddle	75
35	Using an URI-Encoded Querystring to Query Certain Products with Pagination	76

List of Tables

1	Problems in pig supply chains mainly comprising of small backyard operators	24
2	Data Dictionary for Auction	35
3	Data Dictionary for Bid	36
4	Data Dictionary for Buy Order	36
5	Data Dictionary for Location	37
6	Data Dictionary for Certification	37
7	Data Dictionary for Notification	38
8	Data Dictionary for Password	38
9	Data Dictionary for Pig	39
10	Data Dictionary for Location History	39
11	Data Dictionary for Health Record	40
12	Data Dictionary for Weight Record	40
13	Data Dictionary for Feed Nutrition	40
14	Breeding History for Location	40
15	Data Dictionary for Quality Control	40
16	Data Dictionary for Product	41
17	Data Dictionary for Transfer	42
18	Data Dictionary for User	43

I. Introduction

A. Background of the Study

With an early 2022 estimate of about 1.4 million metric tons of pork meat consumed [4], the Philippines ranks among the highest in total meat consumption in the world. Pork accounts for 60% of all meat consumed by Filipinos [5], which was met by a domestic pig production that ranked 9th largest in the world [6] and made up 2.0% to 2.8% of the country's GDP [7]. But despite this seeming enormity, the country is only the 64th largest pig meat exporter in the world in 2020 [8] and has recently experienced sharp inflation in pork prices mainly due to supply deficiencies caused by the 2019 African swine flu [9].

Of the 12.71 million heads in the country's inventory in 2019, 63% were kept by small-scale backyard operators [10]. Small backyard operations tend to have problematic handling of biosecurity and disease control, and they do not benefit from economies of scale [11]. The Pork Producers Federation of the Philippines, Inc. (ProPork) characterizes backyard farming as without investment or planning, and lack of centralization and organization hinders government efforts to lend support to backyard farmers [7].

Despite all this, owing to the fact that still 63% of the country's pork production are from small backyard operators, more focus should be given to them if the country is to achieve self-sufficiency and global exports competitiveness in the pig industry. Organizations like ProPork have shown that it is possible to build reputable brands out of small backyard farms by centralizing their operations, ensuring quality control, and improving their marketing [11].

In light of the devastation caused by the 2019 African swine flu, which killed 40% of the pig population in the country alone since 2019, there is renewed government attention in the livestock industry [12]. Initiatives like eKadiwa, an "online market-

ing platform that directly links producers and agripreneurs to consumers,” is part of ongoing government efforts to modernize and digitalize the agricultural sector. Commercial-scale farms have long already made use of ICT solutions to streamline their operations and maximize their efficiency, but small-scale farmers have yet to benefit from digitalization [13].

Meanwhile, blockchain has been increasingly researched as a candidate technology for the digitalization of supply chains [14]. Blockchain technology is favored for its provision of immutability, data integrity, whole-of-chain transparency, and automation, which are all integral in the modernization of supply chain governance (SCG), or the mechanisms by which supply chains are monitored, materials are sourced, and relationships between supply chain actors are managed, all without compromise to the security and confidentiality afforded by traditional means of SCG in the country. Current applications of blockchain technology to the supply chain do not intend for the technology to be the be-all end-all solution, but for it to work in tandem with other ICT solutions like ERPs, IoT, and other farmer applications.

This paper therefore aims to develop a blockchain-based system of applications and distributed ledger for the digitalization of the Philippine domestic pig industry. In particular, to enhance the supply chain governance of small backyard farms, which are still largely disorganized and without oversight, to streamline processes, and pave the way for reputable branding by way of enhancing quality control through whole-of-chain transparency.

B. Statement of the Problem

An approximate 63% of total pork produce in the Philippines is sourced from small backyard operations [10], where systems for supply chain governance are often lacking. Small backyard operations in the country are characterized by poor handling of biosecurity and disease control [11], and as these farms are typically without reg-

ulatory oversight, the lack of traceability-to-farm capabilities disincentivizes small backyard operators from taking part in the quality control process. Quality checks are not rewarded and the source of contaminated meat is hard to trace.

The implementation of a traceability system that would allow for whole-of-chain transparency and enable consumers and other stakeholders alike to monitor the system would help address these problems and introduce trust and confidence into such supply chains mainly comprising of small backyard operators. However, as there is the risk of data manipulation, the perishability of goods, and the regulatory requirements for pork (e.g., being ASF-free), such a system should also ensure that there is no compromise in security, reliability, and data integrity, where traditional ICT solutions are lacking.

C. Objectives of the Study

C.1 Blockchain System

This research aims to create a blockchain system for the tracking of pigs, portion cuts, and other such information in pig supply chains with a focus on the following properties:

- **Data Immutability** - The blockchain cannot be manipulated to show different data once data is added.
- **Data Integrity** - Data is accurate and consistent throughout the supply chain cycle.
- **Whole-of-Chain Traceability** - The entire process, from breeding to selling, is transparent.
- **Automation** - Certain processes are automated through the system's smart contracts.

C..2 Client Application

This research aims to develop a native mobile client application to interact with the blockchain, and a public API service to provide transparency to the public:

Native Mobile Application

1. Allows the breeders to
 - (a) manage pigs on their breeder farm.
 - (b) view pig information and history at each stage of the supply chain.
 - (c) put pigs for auction at the marketplace.
 - (d) accept/reject bids to their farm's pigs.
 - (e) bid/cancel bid for another farm's pigs.
 - (f) confirm pig transfers into/out of their farms.

2. Allows the raisers to
 - (a) manage pigs on their grower farm.
 - (b) view pig information and history at each stage of the supply chain.
 - (c) put pigs for auction at the marketplace.
 - (d) accept/reject bids to their farm's pigs.
 - (e) bid/cancel bid for another farm's pigs.
 - (f) confirm pig transfers into/out of their farms.

3. Allows the butchers to
 - (a) manage pigs in their slaughterhouse.
 - (b) manage products made from each pig.
 - (c) view pig information and history at each stage of the supply chain.

- (d) bid/cancel bid for another farm's pigs.
 - (e) sell products at the marketplace.
4. Allows the retailers to
- (a) manage products in their store.
 - (b) view pig information and history at each stage of the supply chain.
 - (c) buy products at the marketplace.
5. Allows farm/location managers to
- (a) register new users into their farm/slaughterhouse/store.
 - (b) manage member users of their farm/slaughterhouse/store.
6. Allows the administrators to
- (a) register new users.
 - (b) manage users.

D. Significance of the Project

Traditional means of digitalization would go a long way in making the domestic pork supply chain more efficient, but at the expense of:

- **Security** - Data on traditional databases can be tampered with.
- **Reliability** - Traditional systems can have a single-point of failure.
- **Data Integrity** - Backups are not inherent in traditional databases and can risk unintentional modification or deletion of data.

Meanwhile, a blockchain-based solution has inherent security properties built into its blockchain data structure such that data cannot be tampered with, does not have

a single-point of failure owing to being decentralized, and distributes its database across a peer-to-peer network such that backups are inherent to the system. Aside from this, a blockchain-based solution provides the following, especially in supply chain applications, over other traditional means of digitalization:

- **Immutability** - Supply chain data is transactional. Transactional data records the time, place, price, and other such pertinent data and should not be changeable at any time in the future.
- **Accountability** - Quality control is essential to a reputable brand and especially for those dealing with food. Actors who were negligent in health checks at each stage of the supply chain can easily be held accountable in a blockchain system.
- **Traceability** - Blockchain systems can incentivize quality control in farms as quality and contaminated meat can be traced back to their source farms.

Furthermore, the system's implementation of on-chain and off-chain data storage allows for harnessing of the blockchain ledger's security properties while maintaining the speed and convenience of conventional databases. The project's public API utilizes this to provide transparency and allow for rich queries into the system, providing a platform for other applications, particularly dashboards, to connect to and make use of the supply chain's data.

E. Scope and Limitations

This project

1. only focuses on developing the blockchain-based system and its mobile client application.

2. only considers application to pig supply chains that are mainly comprised of small backyard operators.
3. can only prevent data tampering within the system, where forms of tampering outside the software system are outside the scope of this paper.
4. does not endorse its system as a be-all end-all solution to supply chain governance, and is intended to work in tandem with existing ICT solutions such as ERPs, IoTs, and other farmer applications.
5. does not consider the complexities related to deploying the system in an actual operational setting.
6. does not consider how to get people to actually join the blockchain network.

F. Assumptions

1. That participants, especially those validating transactions, are computer literate, have device/s capable of accessing the necessary applications, and have stable internet access.
2. That malicious actors do not make up a majority of the chain as members at any time.
3. That participants of entire chains, from breeders to retailers, will join the blockchain network at a time.
4. That participants will not buy/sell pigs, or products from/to those that are not part of the blockchain network.

II. Review of Related Literature

The literature is presented in three areas: supply chain governance, overview of blockchain technology, and blockchain technology as applied to supply chains.

A. Supply Chain Governance

[15] defines supply chain governance as the "rules, structures and the institutions that guide, regulate and control the supply chain, emanated from power." Supply chain governance shapes individual and collective actions between supply chain actors as it represents the structures and processes by which they share power. [16] [17]

Two typical supply chain governance mechanisms are identified: (i) contractual governance, which refers to the use of formal contracts to explicitly define each party's responsibilities and obligations, and (ii) relational governance, which refers to the use of trust and relational norms to uphold said responsibilities and obligations [18]. Their study also notes that there is interplay between the two mechanisms and that contracts and trust are (i) independent in cross-border interorganizational relationships (IOR) while being (ii) complementary in vertical IORs (e.g., outsourcing, buyer-supplier relationships).

[1] identifies relational governance as representing the diverse and multi-tiered communicative ecology that is often found in food systems and their supply chains. This proves true in the Philippine setting as well, where studies [7] [2] have shown the frequency of verbal contracts and consignment basis of payment in multiple pig supply chains in the country. One study [2] even noting how an interviewee remarked that *tiwala* or trust is very important in transactions, formal contracts are typically nonexistent for small backyard farms in the country.

Studies also typically only explore IORs and supply chain governance in dyadic relationships, e.g., buyer-supplier and manufacturer-distributor. This study extends

the literature by simulating IORs across an entire multi-tiered pig supply chain modeled from various pig supply chains in the Philippines.

B. Overview of Blockchain Technology

A study in 2021 [14] remarks that blockchain technology is “often considered one of the most remarkable innovations in the 21st century.” The World Bank [19] also defined blockchain as a “novel and fast-evolving approach to recording and sharing data across multiple data stores (or ledgers). This technology allows for transactions and data to be recorded, shared, and synchronized across a distributed network of different network participants.”

The term “blockchain” was first coined in the 2008 whitepaper by the developer/s working under the pseudonym Satoshi Nakamoto. A blockchain can be thought of as a chain of blocks that are sequentially added and ordered and whose copies are distributed across a peer-to-peer (P2P) network such that the nodes or participants in the network can validate each other in place of a central body or another intermediary. [20]

The distributed property of blockchain decentralizes the system, prevents a single-point of failure, and defends the system against attacks such as Denial of Service (DoS) attacks, while the sequentially ordering of blocks with cryptographic hashes that point to the block before them provides the data on the blockchain with immutability (which makes it tamper-resistant/proof) and integrity (which can also be a problem for traditional cloud databases). Blockchain systems make use of a consensus algorithm to form consensus on the network such that a majority of its participants with a copy of the ledger agree on a canonical version of the blockchain, in case that discrepancy arises in a chain’s history of transactions. Bitcoin, the first cryptocurrency, was the first blockchain application. [20]

Blockchain technology has since been studied for application in areas other than

cryptocurrency, including but not limited to: agriculture, carbon market, energy and utilities, fashion, fintech, fish and forest, healthcare, ICT, logistics and supply chain, manufacturing, mining, services, and transportation. [14] noted that the first peer-reviewed papers they were able to scrape were only from 2015, which were back then purely theoretical papers. Their meta-analysis showed an uptrend in volume and increase in proportion of empirical papers, suggesting increasing interest from mainstream industry.

[21] notes the controversies surrounding blockchain technology for its role in cryptocurrencies. Bitcoin, Ethereum, and other such cryptocurrencies have suffered a multitude of hacks, scandals (stemming from both centralized and decentralized practices), problems with regulatory bodies, and other vulnerabilities that erode the people's trust in such systems. [22] however, note that blockchain technology itself worked flawlessly and has successfully been applied to both financial and non-financial world applications.

Nevertheless, blockchain technology only continues to garner more attention for research in every field. Hyperledger Fabric is one such project that acts as framework for developers to easier create entire permissioned blockchain networks and systems of applications benefiting from the properties of blockchain. It was primarily developed by IBM to have a modular and versatile design that allows developers to essentially plug-and-play components such as consensus and membership services to help tailor their system to various industry use cases. The framework has already been used by Walmart and IBM for food traceability and safety, by Hitachi to streamline and secure procurement, by Tech Mahindra to transform Abu Dhabi's land registry, and many more.

C. Blockchain Technology as Applied to Supply Chains

Research into blockchain for use in supply chains is near the forefront of the meta-analysis titled "Is blockchain able to enhance environmental sustainability? A systematic review and research agenda from the perspective of Sustainable Development Goals (SDGs)" [14], making up 11.8% of the papers scraped from 2015 to 2020. Farmer Connect, one of the first applications developed by IBM through Hyperledger Fabric, was also a supply chain solution for traceability and sustainability of the coffee industry. Blockchain technology is also being looked into the shipment industry, where the system's smart contracts is thought to reflect the real-world system and rules into code logic and policy, automating many of the lengthy redundancies of the current supply chain systems.

A study titled "A blockchain-based multisignature approach for supply chain governance: A use case from the Australian beef industry" [1] aimed to digitalize the Australian beef industry, in particular, to apply blockchain technology to a multi-tier and geographically diverse beef supply chain in the country, to address the dyadic relationships between the organizations involved in the supply chain, and to trace (hopefully in real-time) the livestock cycle from breeders to processors. They noted in their literature review that blockchain research into supply chains are many but have mostly been theoretical with no pilot test in the actual setting. Their own research contributed by using a multisignature Proof-of-Authority (PoA) consensus algorithm with the various stakeholders at each stage of the supply chain as validators, where a majority at each stage would validate the evidence and product sent downstream to them in place of computationally expensive consensus algorithms like Proof-of-Work (PoW), enabling tracking of livestock with over 6000 data points in their pilot test with a cost-effective average of \$0.5 USD cost for tracking each livestock.

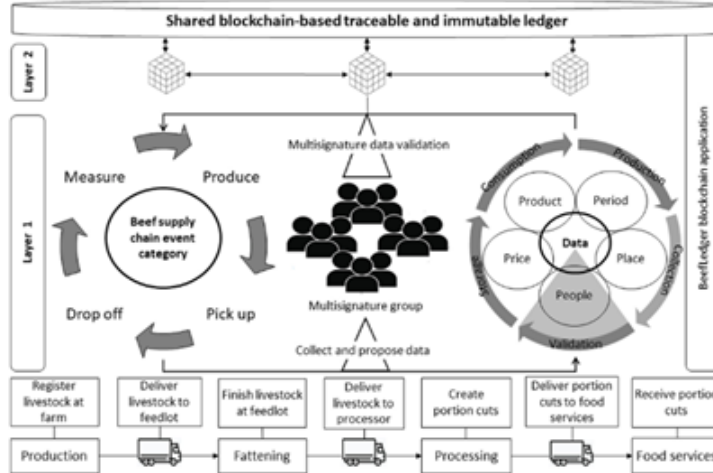


Figure 1: Blockchain-based supply chain governance structure with a multisignature protocol [1]

A study titled "Privacy preserving transparent supply chain management through Hyperledger Fabric" [23] conducted in 2022 simulated the use of the Hyperledger Fabric blockchain framework in a simplified international coffee supply chain, addressing the issues of traditional systems of non-transparency, monopoly, asymmetry, susceptibility to tampering, and single-point of failure, while also addressing the issues of public blockchain implementations of limited scalability, lack of anonymity, modularity, and so on.

It is not a silver bullet for all supply chains though. The paper "Blockchain is not a silver bullet for agro-food supply chain sustainability: Insights from a coffee case study" [24] noted the incompatibilities of blockchain technology with the current infrastructure of the coffee supply chain in their case study. Coffee cherries tended to be mixed in larger mills early in the chain rendering traceability to farm, a core advantage of blockchain technology in supply chains, moot. Although they remarked that the technology can be applied for niche coffee.

A study on the effects of blockchain technology on supply chains notes that trust is shown to be a strong measure of a supply chain's success. Information sharing

between organizations in the supply chain is often limited such that demand variation increases upstream and lead time variation increases downstream. They compare a simulated blockchain implementation's performance compared to previous blockchain and non-blockchain solutions (IoT and big data, conventional methods), noting that through the facilitation of trust, information sharing, and automation through the blockchain, costs incurred due to the bullwhip effect by key actors in the supply chain can go down (in their simulation) by half or to even a fifth of the original cost. [25]

Solutions through blockchain in other industries can play a role in the supply chain as well. A study titled "Secure decentralized electronic health records sharing system based on blockchains" [26] used blockchain technology to secure electronic health records while providing patients privacy of information (even to staff that are unrelated to his/her case) and ownership of their information (such that they can sell or grant access to their data to researchers). The study also made use of Interplanetary File System (IPFS) to store hashed or encrypted data offchain for better performance while preventing a single point of failure (IPFS is distributed). Key actors in the supply chain (especially smallholders like the producers) may be granted privacy and ownership of their data in the same way.

Another study titled "Blockchain-based mobile crowdsourcing model with task security and task assignment" [27] also used blockchain technology for crowdsourcing where the whole process from task assignment to payment (and punishment) is automated. This can also be applied to supply chain systems such that supply and demand are automatically matched together.

C.1.1 Synthesis

Supply chain governance, or the mechanisms which govern the rules and transactions in a supply chain, shapes the individual and collective actions of the supply chain actors and is an important factor in the cohesiveness and efficiency of a supply chain

as a whole. Blockchain technology, while having been first used for finance technology, i.e., cryptocurrencies, have since been used in many other fields including but not limited to: energy and utilities, transportation, the carbon market, and logistics and supply chain. Supply chain application is one of the most popular use-cases for blockchain technology, in particular supply chain governance, as the technology's inherent provision of security, reliability, and data integrity are important for supply chains.

Many papers have since come to explore the use of blockchain technology in supply chain applications, in particular, to extend the dyadic relationships between supply chain actors, to address the *bullwhip effect* (BWE) by coordinating information sharing across the chain, to incentivize quality control and sustainable practices by enabling traceability to farm, and many more. Blockchain research into supply chain applications have also recently seen more empirical papers, with pilot tests having been done in a multi-tier and geographically diverse beef supply chain in Australia, and a complex multinational coffee supply chain involving a cooperative of coffee growers from Antioquia, Colombia.

The existing literature, though, have only tested on supply chains involving large and established organizations. Few papers, if any, have explored the benefits of blockchain technology to smaller supply chains involving smaller actors, like the backyard pig farmers in the Philippines. This paper contributes to the existing literature by exploring the benefits and a possible implementation for a blockchain system to regulate the supply chain governance of smaller supply chain actors.

III. Theoretical Framework

A. Overview of Pork Supply Chains in the Philippines

Pig production is a dominant activity throughout the country. A study by Ayomen and Kingan (2019) in particular noted an age range of 21 to 68 for the pig raisers in the highlands of Sablan, Benguet [7], with several other studies noting similar age ranges in different areas of the country [2] [3]. Fig. 2 illustrates the relationship dynamics between the supply chain actors for lechon processing in Quezon Province.

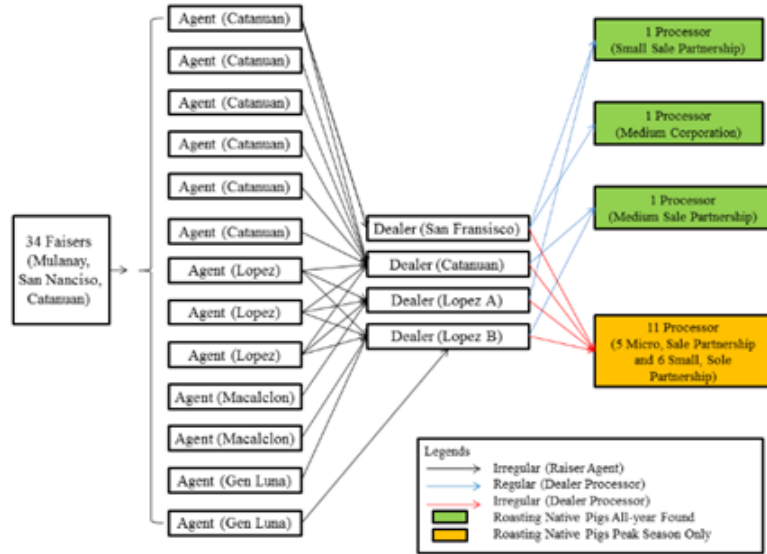


Figure 2: Relationship Dynamics between Value Chain Actor Respondents [2]

B. Blockchain Technology

Blockchain technology generally refers to systems that make use of a ledger database distributed across a network of peers where computers add and validate transactions on a *linked list* or chain of blocks. Generally, blockchains are comprised of:

- **Distributed Ledger** - The record of all transactions.

- **Peer-to-Peer Network (P2P)** - Computers or *nodes* that are linked in a clustered way and work independently to write and validate *blocks* of transactional data.
- **Consensus Mechanism** - The process by which the nodes on the blockchain network achieve consensus, i.e., agree on a single valid chain of transactions.
- **Incentive Mechanism** - The mechanism(s) by which nodes are incentivized to be "honest" in writing or validating blocks.

C. Hyperledger Fabric

The Hyperledger Fabric is a blockchain framework developed by IBM, and is intended for developers to easier create entire permissioned blockchain networks and systems of applications benefiting from the properties of blockchain. The framework was developed to have a modular and versatile design that enables developers to essentially plug-and-play components such as consensus and membership services to help tailor their system to various specific industry use cases. The framework has already been used by Walmart and IBM for food traceability and safety, by Hitachi to streamline and secure procurement, by Tech Mahindra to transform Abu Dhabi's land registry, and many more. Projects developed through the Hyperledger Fabric generally comprise of the following:

- **Identity** - Each actor, whether they be peers, orderers, or even administrators, are identified in the blockchain network by an X.509 digital certificate. This is the most common type of digital certificate and includes a public key, digital signature, and information regarding both its user and the *Certificate Authority* (CA) that issued it.
- **Membership Service Provider (MSP)** - Is responsible for converting identities into roles for interacting with the blockchain network.

- **Policies** - Are the set of rules defining how decisions are made and how certain outcomes are reached for certain operations in the blockchain network. In particular, Access Control Lists (ACLs) define how resources, e.g., chaincode, are accessed, smart contract endorsement policies define how many *endorsements* from endorsing peers are required for transactions to be considered valid, while modification policies define the identities required to approve any configuration update, including those updating certain policies, for the update to actually push through.
- **Peers** - Also referred to as peer nodes, are the nodes that host and manage a copy of the ledgers and smart contracts of the blockchain network. As of Hyperledger Fabric v2.4, certain peers, called the endorsing peers, are responsible for the independent verification of transaction proposals by client applications and the subsequent endorsement of the transaction if they are to pass certain checks. A certain threshold for endorsements are required for transactions to pass, be ordered by orderers into blocks, and included by all peers in their own ledger database, as defined by the relevant endorsement policies.
- **Ledger** - Contains both the current state and historical data of transactions and distributed to all peers on the blockchain network.
- **Ordering Service** - Are formed by orderers, or orderer nodes, which order the transactions into blocks which are then sent to all peers for updating of their own hosted ledger database. Whereas the consensus mechanism in Hyperledger Fabric is performed by its endorsing peers and is deterministic, i.e. peers are guaranteed to reach a consensus on a single canonical valid chain without *forks*, the ordering service that comes after the consensus mechanism is also final and correct. This is because nodes on the blockchain network in Hyperledger Fabric do not compete with each other for incentives, instead, validation of transaction

proposals are delegated to peers in policy files.

- **Smart Contracts and Chaincode** - Smart contracts define the executable logic that generate new data that are added to the database ledger. Related smart contracts are grouped into chaincodes for deployment.
- **REST API** - REST APIs, meaning, Representational State Transfer (REST) Application Programming Interface (API), are APIs that conform to the REST architectural style set of constraints that allow for e.g., web services to interact with one another. Client applications interact with the blockchain through the REST API layer, which expose certain endpoints for the user to invoke chaincode(s) on the blockchain network to submit their transaction proposals.

D. Apache CouchDB

Whereas LevelDB is the default state database for Hyperledger Fabric projects where chaincode data is stored as simple key-value pairs and where only key queries are supported, CouchDB is the alternate state database that allows for the modeling of data as JSON and supports rich queries that allow for data to be queried by value instead of their key.

E. Azure

Azure is a cloud computing platform run by Microsoft. As development on Hyperledger Fabric requires a Linux-based OS, and its nodes are resource intensive, peer nodes in particular requiring at least 1gb of memory allocated each, Azure's **B4ms** series virtual machine, specialized for high memory needs, was used for developing the system. The virtual machine runs Ubuntu 20.04.

F. Docker

Docker is a set of Platform as a service (PaaS) products that uses OS-level virtualization to package software in packages called containers, providing an efficient way to create, deploy, and run applications. Containerization eases deployment of the same application across different environments, from development to production, without worrying about compatibility issues.

Docker is commonly used to package and deploy Hyperledger Fabric nodes and other components, allowing organizations to easily spin up new nodes, manage them, and scale up or down as necessary.

G. Kubernetes

Kubernetes is a container orchestration system that automates the deployment, scaling, and management of containerized applications. It provides a way to manage and orchestrate multiple Docker containers across multiple hosts, making it easier to manage large-scale distributed applications.

Kubernetes is often used with Hyperledger Fabric and Docker to manage and orchestrate the deployment of Fabric nodes and services across multiple hosts, which becomes necessary as blockchain networks get larger. This helps simplify the deployment and management of Fabric networks and improve scalability and availability.

H. Hyperledger Caliper

Hyperledger Caliper is a blockchain benchmarking tool for evaluating blockchain implementations in terms of certain performance metrics, such as success rate, transaction & read throughput, transaction & read latency, and resource consumption.

Hyperledger Caliper measures the system's performance using a set of predefined use cases.

IV. Design and Implementation

As the paper’s objectives are to provide a blockchain-based solution to digitalize the supply chain governance in domestic pork supply chains, an exploratory case study approach is adopted to both guide the implementation of and evaluation of the blockchain-based system.

A. Blockchain Pork Supply Chain

To achieve the paper’s objective of a blockchain-based solution for supply chain governance in pig supply chains mainly involving smallholders in the form of small backyard operators, the specific supply chain structure used in the system is modeled around existing supply chain(s) that are comprised of said operators. For this, we look at three studies surveying the respective pork supply chains in various areas across the country:

- Sablan, Benguet (highlands) (2019) [7]
- Quezon Province and La Loma, Quezon City (lowlands) (2014) [2]
- Western Leyte (lowlands) (2002) [3]

Of the three studies, the supply chain explored in the Leyte study painted the most diverse channels for pigs to flow from producer to end consumer:

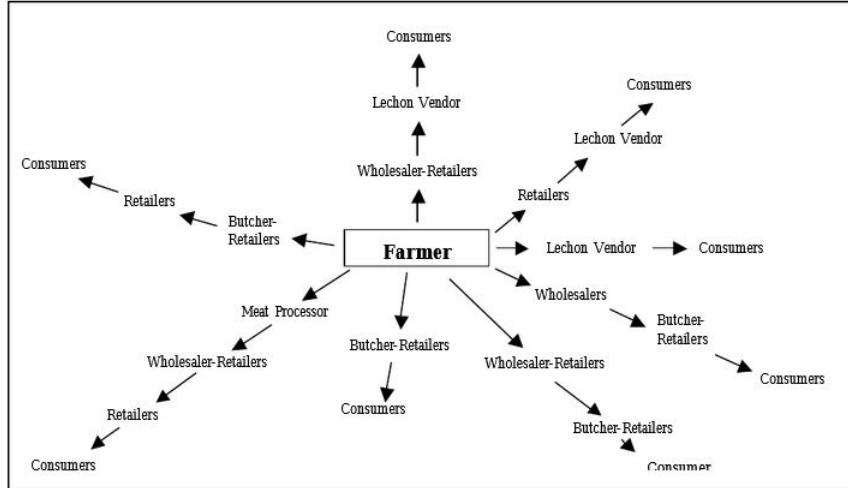


Figure 3: Pork supply chain in Western Leyte [3]

This study likely paints the most accurate depiction of a domestic pork supply chain in terms of diversity, but the study notes that the usual channel for producers to sell their pigs to is "farmer-wholesalers-butcher/retailers-consumer". This usual channel matches with the channels in Sablan, Benguet, and Quezon Province, where pigs are generally transferred as follows: farmer-traders-vendors/processors.

Ayomen et al. (2019) noted that their study in Benguet coincided with the study in Quezon Province in that producers tended to only sell to traders or retailers. The pig raisers in Leyte had the greatest flexibility in that they sold to everyone downstream, including end consumers.

Wholesalers (or traders) present the largest complexity in the supply chain, as they are essentially transporters, and therefore distributors who transport products from butcher to retailers can also be regarded as traders. The study by Cao et al. (2022) in Australia also did not include the transporters as key actors in their system [1]. Hence, traders are not included as users in this system, and the key supply chain actors that are included as users in the system therefore can be categorized as follows: breeders, raisers, butchers, and retailers.

The 4 key actors are described as follows:

1. **Breeders** - breeds and farrows sows and keeps nursing piglets until they are sold as feeder pigs to raisers. Breeders can also buy sows from other breeders.
2. **Raisers** - buys feeder pigs from breeders and grows them to market weight.
3. **Butchers** - slaughters and processes pigs into products and sells to retailers.
4. **Retailers** - sells products or processed pig meat products to end consumers.

B. Use Case Design

B.1 Requirement Analysis

The system's design requirements take into factor the structure of the system's pork supply chain and the circumstances in the aforementioned studies' pork supply chains. As computer literacy, availability of internet-capable devices, and stable internet connections are part of the paper's assumptions, the circumstances here refer to the problems painted by the studies regarding their areas' respective supply chains. These problems are collated as follows:

Problem	Details
Flow of information is lacking	Members do not know the state of the entire chain and only hear from members adjacent to them in the chain. Lack of information causes imbalanced negotiating powers and mismatch between supply and demand.
Lack of contracts in transactions	Payments are informal and can be on a consignment basis. Nonpayment happens at times especially for the producers, hence why the members interviewed remarked that "trust" is very important.
Limited market reach for producers	Studies have shown that certain neighboring areas do not know of excess supply or demand in other areas. Members often only transact with other members in their area.
Limited financial capability of members	Funds and profit are low especially for producers. Private investment and government intervention are lacking.
Imbalanced negotiating power favoring downstream actors	Studies have shown that (lechon) processors and (wholesale) traders dictated the price and payment for pigs. Their profits do not necessarily translate to profits for producers.

Table 1: Problems in pig supply chains mainly comprising of small backyard operators

Basing on the above problems and other circumstances, 4 key strategic points are identified for the blockchain-based solution to have:

1. easy application for low tech actors
2. whole-of-chain transparency for free flow of information

3. implementation of supply chain transactions in smart contracts
4. does not charge any actor for its services

B.2 Design and Development

To design the system prototype, the risks and disadvantages to blockchain, especially as applied to supply chain purposes, are identified as follows:

1. **Garbage in/Garbage out** - the system needs a way to filter out malicious/problematic inputs (and malicious users).
2. **Data Privacy** - the chain's data, including personal and other confidential information, are duplicated in ledgers distributed to other members of the chain.
3. **Cost-Effectiveness** - blockchain implementations (e.g., cryptocurrency) have a history of being expensive to maintain, this including the consensus mechanism and the data storage problem.
4. **Honesty of Members** - malicious actors should not make up the majority of the network's members at any time.

To address the above, the following solutions are implemented in the system:

1. **Proof of Authority-based (PoA) Multisignature Architecture** - information regarding transactions between two or more members need the tacit approval of all members involved as validation.
2. **Private Blockchain** - private blockchains differ from public blockchains in that membership requires the approval of its members (and that members' identities are known), this simplifies the security mechanisms needed for the network and greatly reduces network traffic.

3. **Deterministic Consensus Algorithm** - deterministic consensus algorithms differ from probabilistic consensus algorithms (used in public blockchains) in that they do not need to provide incentives for validators since the members of the private blockchain do not compete with each other and trust each other (enough to permit them joining the chain), this also means a much faster finality of transactions and much less computing costs.
4. **Permissioned Blockchain** - permissioned blockchains differs from permissionless blockchains in that the network has rules on who have rights/authority to access certain data, transact, and so on.

Fig. 4 illustrates the standard flow of pigs and products throughout the supply chain, where: (i) the breeders would sell feeder pigs to raisers, (ii) the raisers would sell market pigs to the butchers, (iii) the butchers would slaughter and process the market pigs into products and sell to retailers, (iv) and retailers would finally sell the products to the end consumers.

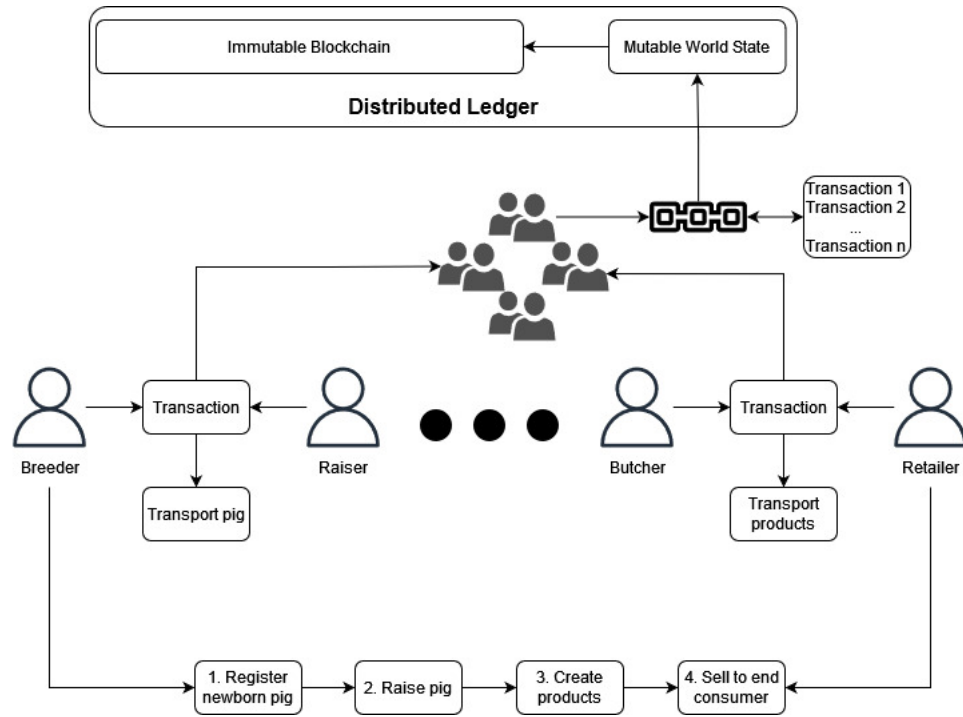


Figure 4: PoA-based Multisignature pork supply chain architecture

The implemented supply chain system, however, offers increased flexibility by enabling breeders to engage in pig selling and transfers among themselves. Additionally, breeders are also given the option to purchase pigs from raisers. This enhanced flexibility aims to provide greater autonomy to farm locations in managing their pig population. It allows breeders to address various situations such as shortages of sow pigs or the temporary need for boars, ensuring efficient pig distribution and supporting the unique requirements of each farm.

Guided by the aforementioned solutions, the system is developed through Hyperledger Fabric, an open source enterprise-grade permissioned distributed ledger technology (DLT) platform developed by IBM to boost development of custom enterprise blockchain solutions. Hyperledger Fabric was chosen for its highly modular and configurable architecture, where components such as consensus and membership services are plug-and-play.

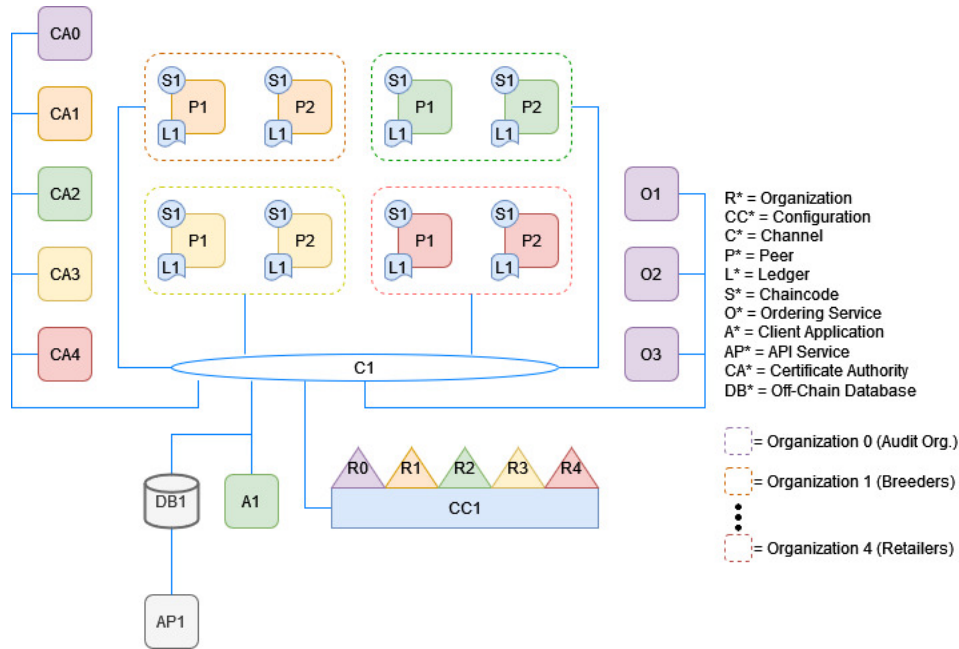


Figure 5: The blockchain network

Visualized in Fig. 5 is the blockchain network for the pig supply chain. There are in total 5 organizations, consisting of an orderer organization that is handled by an external audit body, and 4 peer organizations (breeders, raisers, butchers, and retailers), each with two peer nodes for redundancy, where each peer hosts an instance of the same ledger L1, which contains both the mutable world state and blockchain, and the same chaincode S1. Participants become members in each organization through their organization’s own certificate authority CA. Administrators are the one to register new users and will give them their public-private key pair and X.509 certificate for their identity.

All organizations are connected to a single channel C1 to facilitate transactions with each other, where C1 is configured in configuration CC1, detailing the authority of each organization and the endorsements/signatures necessary to approve each transaction.

A1 is a native mobile application that would render the relevant pages to the user

depending on their role (e.g., raiser), and has the functionalities for writing data to the blockchain. Meanwhile, AP1 is a public API for the public (including members) to perform rich queries into the system with the speed of a conventional off-chain slave database DB1, which serves as a copy of the mutable world state and hosted outside the blockchain network.

Members of each organization can share data with each other (transaction proposals, etc.) through the channel C1. Members would access their application A1 to access the network's REST API layer through an anchor peer, which abstracts the chaincode S1. It is the chaincode S1's smart contracts that actually has the ability to append blocks of data to the blockchain part of L1, and update the data (state database or mutable world state) of the same ledger L1.

The blockchain part of L1 is the sequential chain of blocks of transactions that form the historical data of the network and provide its traceability features. The state database of L1, much like those in Ethereum, houses the latest data of the network. This distinction allows the network to have a complete record of the history of transactions through the blockchain, while not requiring a full retracing of the blockchain just to extract the latest data by having a separate state database (unlike e.g., Bitcoin), allowing for a higher throughput without compromising on traceability.

We define 3 node types necessary to the cycle of transactions in the blockchain network:

- **Anchor Peer** - a single peer in each organization that facilitates sending of data through the channel to other organizations.
- **Endorsement Peer** - peers who have endorsement capabilities, where a certain number of which is necessary for a transaction to be passed to the ordering service.
- **Orderer Node** - nodes that order the transactions sent to them in blocks for

committing to the blockchain.

We detail the cycle of each transaction as follows. Note that much of this is abstracted away and from the user's perspective, they are interacting with a regular mobile app.

1. **Transaction proposal** - A member would propose a transaction by requesting to invoke a chaincode S1, which the member would sign and submit.
2. **Transaction endorsement** - The endorsing peers would verify the signature on the transaction and perform a preliminary check on it (authority, authenticity, etc.). If all checks pass, the transaction is executed on the peer. If nothing fails, the values produced by the chaincode execution are signed by the endorsing peers (they're working independently) and sent back to the proposing member as endorsement.
3. **Collection of endorsements and ordering request** - The proposing member checks the received endorsements against the network policies. If the transaction was a write operation, the proposing member requests for the orderers O1 to O3 to process their transaction to append to the blockchain. If the transaction was only a read operation, no ordering process is requested.
4. **Transaction validation and commit** - The orderers distribute the ordered transactions to all peers, which they would independently validate and commit to their ledger if all checks pass.

To facilitate ease of use of the applications for low tech actors, the functionalities of the application A1 is akin to other conventional applications, that is, the UI/UX are similar to that of conventional mobile applications despite the use of blockchain technology in the system. Aside from the certificates necessary to log in, there is little difference between the client application A1 and other mobile applications.

C. Use Cases

Below are the use case diagrams for the PorkWatch client mobile application, and two activity diagrams illustrating the login process and the interaction between the main server and the blockchain during transaction processing, both of which are prevalent in all of the use cases of the mobile app. The specific use cases of the PorkWatch mobile application are already outlined in the Objectives of the Study section of this paper.

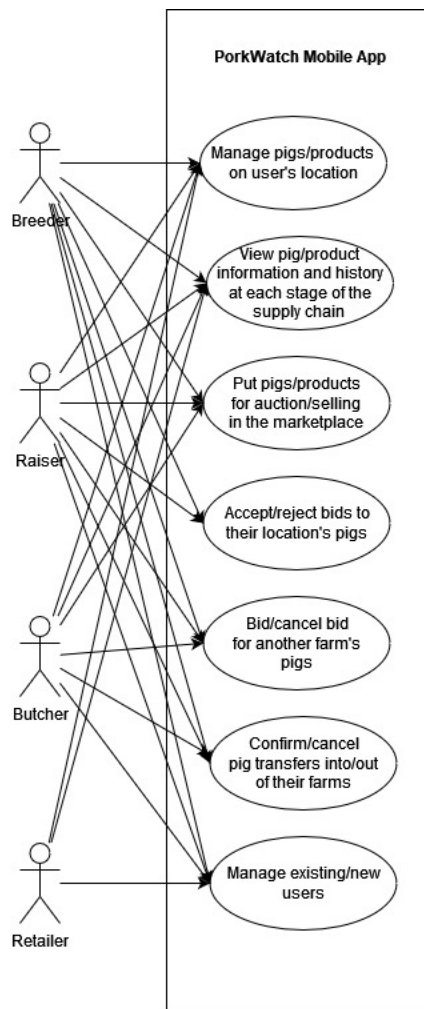


Figure 6: Use Case Diagrams for the PorkWatch client mobile application

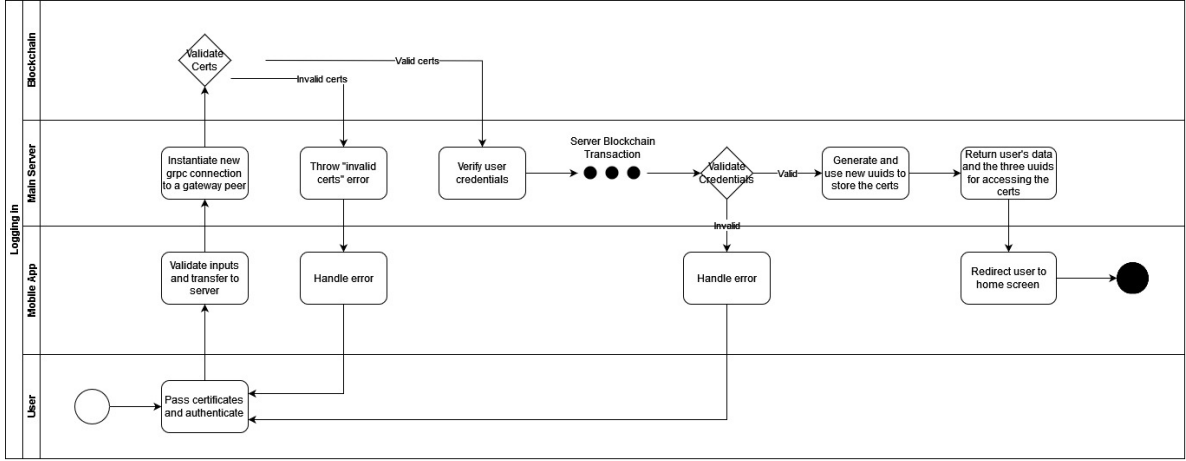


Figure 7: Activity Diagram for Logging in to the Mobile App

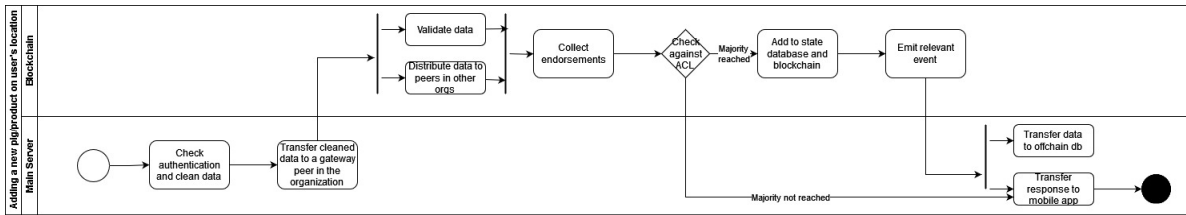


Figure 8: Activity Diagram for the Server-Blockchain Transaction Process

D. Database Design

The system makes use of an on-chain state database that is replicated across all peer nodes in the blockchain network, and a single off-chain database deployed outside the blockchain network. The on-chain state databases act as masters where logged-in users could read/write to, and are the only databases that can and will handle blockchain transactions in the network. The off-chain database act as a slave database, whose main purpose is to keep up-to-date with the on-chain master databases and be read from by all users, logged-in or not.

This serves to limit needless traffic in the blockchain network, as users of the dash-

board, who do not need to log in, can view data from the off-chain slave database. Both types of databases are document-oriented NoSQL CouchDB databases.

The slave database is meant to replicate much of the on-chain master databases, but with the exemption of the password documents, and thus share much of the same data model and data dictionary, as shown below in that order. Note that nested objects were represented also as collections in the data dictionary, but can be identified by their lack of an **Id** field.

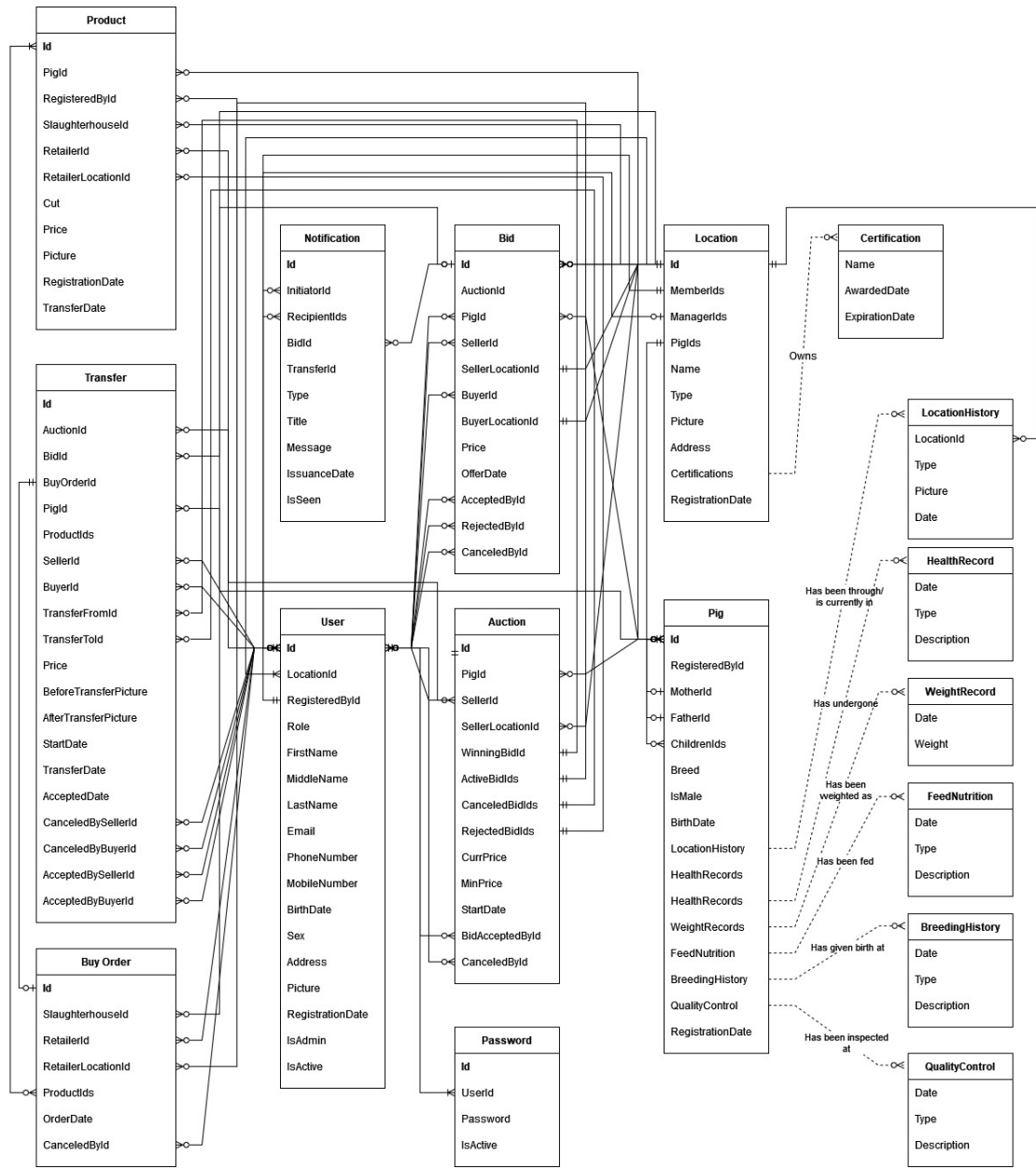


Figure 9: Data Model

Field Name	Field Type	Field Description	Nullable
Id	String	Unique Identifier for the auction document	No
PigId	String	References the pig the auction is for	No
SellerId	String	References the user that initiated the auction	No
SellerLocationId	String	References the location of the user when the auction was initiated	No
WinningBidId	String	References the bid that won the auction	No
ActiveBidIds	String[]	References the bids for the auction that are neither canceled nor rejected	No
CanceledBidIds	String[]	References the bids for the auction that have been canceled	No
RejectedBidIds	String[]	References the bids for the auction that have been rejected	No
CurrPrice	Float	The price of the highest active bid on the auction	No
MinPrice	Float	The minimum price allowed for bids on the auction	No
StartDate	String	The date when the auction was initiated	No
BidAcceptedById	String	References the user that accepted the winning bid, if any	Yes
CanceledById	String	References the user that canceled the auction, if any	Yes

Table 2: Data Dictionary for Auction

Field Name	Field Type	Field Description	Nullable
Id	String	Unique Identifier for the bid document	No
AuctionId	String	References the auction the bid is for	No
PigId	String	References the pig the bid is for	No
SellerId	String	References the user that initiated the auction the bid is for	No
SellerLocationId	String	References the location of the selling user when the auction was initiated	No
BuyerId	String	References the user that initiated the bid on the buyer side	No
BuyerLocationId	String	References the location of the user when the bid was initiated	No
Price	Float	The price of the bid	No
OfferDate	String	The date when the bid was initiated	No
AcceptedById	String	References the user that accepted the bid	Yes
RejectedById	String	References the user that rejected the bid	Yes
CanceledById	String	References the user the canceled the bid	Yes

Table 3: Data Dictionary for Bid

Field Name	Field Type	Field Description	Nullable
Id	String	Unique Identifier for the buy order document	No
SlaughterhouseId	String	References the slaughterhouse where the products being bought are from	No
RetailerId	String	References the retailer that initiated the buy order	No
RetailerLocationId	String	References the location of the retailer that initiated the buy order	No
ProductIds	String[]	References the products that the buy order pertains to	No
OrderDate	String	The time when the buy order was initiated	No
CanceledById	String	References the user on the buying side that canceled the buy order, if any	No

Table 4: Data Dictionary for Buy Order

Field Name	Field Type	Field Description	Nullable
Id	String	Unique Identifier for the location document	No
MemberIds	String[]	References the Ids of the users that are currently members of the location	No
ManagerIds	String[]	References the Ids of the users that are currently acting as managers of the location (in the app)	No
PigIds	String	References the pigs that are currently owned by the location	Yes
Name	String	The name of the location (in the app)	No
Type	String	The type of the location, i.e., Breeder, Raiser, Butcher, Retailer	No
Picture	String	A recent picture of the location	No
Address	String	The registered address of the location	No
Certifications	Certification[]	Consists of the certifications the location has registered	Yes
RegistrationDate	String	The date when the location was registered in the app	No

Table 5: Data Dictionary for Location

Field Name	Field Type	Field Description	Nullable
Name	String	The name of the certification	No
AwardedDate	String	The date when the certification was awarded	No
ExpirationDate	String	The date when the certification will expire	Yes

Table 6: Data Dictionary for Certification

Field Name	Field Type	Field Description	Nullable
Id	String	Unique identifier for the notification document	No
InitiatorId	String	References the user that caused/triggered the notification to be created and distributed	No
RecipientIds	String[]	References the user/s that should receive the notification	No
BidId	String	References the bid the notification is about	No
TransferId	String	References the transfer the notification is about	No
Type	String	The type of the notification, e.g., Bid, Transfer, Member Addition/Removal	No
Title	String	The title of the notification	No
Message	String	The message/body of the notification	No
IssuanceDate	String	The date when the notification was issued	No
IsSeen	Boolean[]	Is true if the respective user has already seen the notification	No

Table 7: Data Dictionary for Notification

Field Name	Field Type	Field Description	Nullable
Id	String	Unique identifier for the password document	No
UserId	String	References the user that owns the password	No
Password	String	The hash of the password	No
IsActive	Boolean	Is true if the password is active, i.e., if the user can log in using it (actions like changing the password changes this)	No

Table 8: Data Dictionary for Password

Field Name	Field Type	Field Description	Nullable
Id	String	Unique identifier for the pig document	No
RegisteredById	String	References the user that registered the pig first	No
MotherId	String	References the pig's mother	Yes
FatherId	String	References the pig's father	Yes
ChildrenIds	String[]	References the pig's children	Yes
Breed	String	The breed of the pig, e.g., Duroc Pig, Hampshire Pig	No
IsMale	Boolean	Is true if the pig is male, false is female	No
BirthDate	String	The date when the pig was birthed	No
LocationHistory	LocationHistory[]	Consists of the locations the pig has been through/currently residing in	No
HealthRecords	HealthRecord[]	Consists of the health checks the pig has undergone	Yes
WeightRecords	WeightRecord[]	Consists of the measured weights of the pig each time they're measured	Yes
FeedNutrition	FeedNutrition[]	Consists of the feed the pig has been eating	Yes
BreedingHistory	BreedingHistory[]	Consists of descriptions of when the pig has birthed	Yes
QualityControl	QualityControl[]	Consists of quality checks the pig has undergone	Yes
RegistrationDate	String	The date when the pig was first registered	No

Table 9: Data Dictionary for Pig

Field Name	Field Type	Field Description	Nullable
LocationId	String	References the location of the pig at this instance	No
Type	String	The type of the location, i.e., Breeder, Raiser, Butcher, Retailer	No
Picture	String	The picture of the location at this instance	No
Date	String	The date when the pig was transferred to this location	No

Table 10: Data Dictionary for Location History

Field Name	Field Type	Field Description	Nullable
Date	String	The date when the health record was taken	No
Type	String	The type of the health check	No
Description	String	A description of the health check	No

Table 11: Data Dictionary for Health Record

Field Name	Field Type	Field Description	Nullable
Date	String	The date of this instance	No
Weight	Float	The weight during this instance	No

Table 12: Data Dictionary for Weight Record

Field Name	Field Type	Field Description	Nullable
Date	String	The date when the record was taken	No
Type	String	The type of the feed	No
Description	String	A description of the feed	No

Table 13: Data Dictionary for Feed Nutrition

Field Name	Field Type	Field Description	Nullable
Date	String	The date of this instance	No
Type	String	The type of the breed	No
Description	String	A description of this instance	No

Table 14: Breeding History for Location

Field Name	Field Type	Field Description	Nullable
Date	String	The date when the quality control check took place	No
Type	String	The type of the quality check	No
Description	String	A description of the quality check	No

Table 15: Data Dictionary for Quality Control

Field Name	Field Type	Field Description	Nullable
Id	String	Unique identifier for the product document	No
PigId	String	References the source pig of this product	No
RegisteredById	String	References the butcher that registered the product	No
SlaughterhouseId	String	References the slaughterhouse where the product was registered	No
RetailerId	String	References the retailer that bought the product	No
RetailerLocationId	String	References the retail location where the product was sold to	Yes
Cut	String	Describes the cut of meat, if applicable	Yes
Picture	String	A picture of the product upon registration	No
RegistrationDate	String	The date when the product was registered	No

Table 16: Data Dictionary for Product

Field Name	Field Type	Field Description	Nullable
Id	String	Unique identifier for the transfer document	No
AuctionId	String	References the auction the transfer is related to	Yes
BidId	String	References the bid the transfer is related to	Yes
BuyOrderId	String	References the buy order the transfer is related to	Yes
PigId	String	References the pig the transfer is about	Yes
ProductIds	String	References the products the transfer is about	Yes
SellerId	String	References the user that accepted the winning bid or sold the product	No
BuyerId	String	References the user that initiated the winning bid or bought the product	No
TransferFromId	String	References the location of the selling user when the pig or product was sold	No
TransferToId	String	References the location of the buying user when the pig or product was sold	No
Price	Float	The price with which the pig/product was successfully bought	No
BeforeTransferPicture	String	The picture of the pig before the transfer has commenced	Yes
AfterTransferPicture	String	The picture of the pig after the transfer has finished	Yes
StartDate	String	The date when the transfer has initiated	No
TransferDate	String	The date when the transfer has actually commenced	Yes
AcceptedDate	String	The date when the transfer has actually finished	Yes
CanceledBySellerId	String	References the user on the selling side that canceled the transfer	Yes
CanceledByBuyerId	String	References the user on the buying side that accepted the transfer	Yes
AcceptedBySellerId	String	References the user on the selling side that accepted the transfer	Yes
AcceptedByBuyerId	String	References the user on the buying side that accepted the transfer	Yes

Table 17: Data Dictionary for Transfer

Field Name	Field Type	Field Description	Nullable
Id	String	Unique identifier for the user document	No
LocationId	String	References the location the user is currently a member of	No
RegisteredById	String	References the user that registered this user into the app	No
Role	String	The role of the user, i.e., Breeder, Raiser, Butcher, Retailer	No
FirstName	String	The first name of the user	No
MiddleName	String	The middle name of the user	Yes
LastName	String	The last name of the user	No
Email	String	The unique registered email of the user	No
PhoneNumber	String	The registered phone number of the user	Yes
MobileNumber	String	The registered mobile number of the user	Yes
BirthDate	String	The date when the user was born	No
Sex	String	The sex assigned at birth of the user, i.e., Male or Female	No
Address	String	The registered address of the user	No
Picture	String	A recent picture of the user	No
RegistrationDate	String	The date when the user has/was first registered in the app	No
IsAdmin	Boolean	Is true if the user is an admin in the app	No
IsActive	Boolean	Is true if the user is currently active in the app	No

Table 18: Data Dictionary for User

E. System Architecture

The system comprises of a Hyperledger Fabric production network, a main backend server, a native mobile client application, a public API server, and an off-chain slave database with the following technology stack:

- **React** - Frontend UI Framework
- **Redux** - Frontend State Management Library
- **React-Bootstrap** - Frontend CSS Framework
- **Node.js** - Server Environment

- **Express** - Backend Framework
- **WSL2** - Compatibility Layer for Linux
- **Docker** - Containerization Platform
- **Hyperledger Fabric** - Blockchain Framework
- **Apache CouchDB** - Document-oriented NoSQL Database
- **Microfab** - Containerized Hyperledger Fabric runtime
- **Kubernetes** - Container Orchestration System

The blockchain network is built using Hyperledger Fabric and Apache CouchDB, and serves as the core of the system's data storage and management capabilities. It consists of multiple nodes, each node managed by an organization, that interact with each other to maintain a distributed ledger of transactions and smart contracts.

Both the main backend server and the public API server are built using Node.js/Express. The main backend server provides an API service for the mobile app to interact with the blockchain network and retrieve data from the on-chain databases. The main server implements business logic for the system, including user authentication, transaction processing, and data validation. Additionally, the main server listens to events emitted on the blockchain, processes their payload, and handles inserting them to the off-chain slave database for the public API to use.

The mobile app is built using React Native Expo. It provides a user interface for the supply chain's key actors, i.e., the breeders, raisers, butchers, and retailers, for interacting with the system. It communicates with the backend server via REST API calls, and uses a combination of local and remote data storage to manage user data and state.

The project's system architecture is designed to be modular and scalable, owing to Hyperledger Fabric's modular design and the use of Kubernetes pods and docker

containers for hosting each blockchain component. The justfiles and shell files can be promptly edited to modify the number of organizations and their access control, while the .yaml and .yml files, and Dockerfiles can be edited to modify the resource allocation for the docker containers.

The blockchain and servers have been developed in and are deployed in production in an Azure **B4ms** series virtual machine with Ubuntu 20.04.6 LTS as OS, a 64gb premium SSD managed disk. The mobile app was simultaneously developed on the local WSL2, using either a AMD Ryzen 5 3400GE desktop or a Ryzen 5 2500U laptop, and deployed on Expo as an android build.

F. Technical Architecture

Running the full system, from blockchain to frontend, requires at least:

- **Processor:** At least more than 3 cpu units minimum; 4 cpu units recommended
- **Memory:** 16gb minimum
- **Storage:** 64gb minimum
- **Screen Resolution:** >300 px width or larger; any height
- **Internet Connection:** required

Although running the system on only 3 cpu units has not been tested, a minimum of 3 cpu units is necessary as the sole worker node for the kubernetes cluster requires a minimum of 2800 milliCPU units. 4 processors is recommended as this might ignore spikes in usage and may fail in practice.

The blockchain network contains a single channel with 8 peer nodes, 3 orderer nodes, and 5 certificate authorities. Their minimum memory requirements are as follows, for a total of 13.7gb of memory:

- **Peer Node:** 1.25gb
- **(3x) Orderer Nodes:** 1.6gb
- **Certificate Authority:** 420mb

The above were estimated from their respective yaml configuration files. The Hyperledger Fabric documentation actually recommends allocating 1gb of memory for the peer itself, which is just a part of the peer node yaml configuration file. Allocating 1gb of memory for the peer itself will increase the memory usage of each peer node to 1.5gb, increasing the total memory usage of the blockchain network to 15.7gb. To limit costs, each peer is only allocated 75% of the recommended memory unit at 750mb.

Azure's **B4ms** series virtual machine was unable to handle the project with only a 32gb managed disk, therefore a minimum of 64gb of storage is deemed minimum for running the blockchain locally.

A stable internet connection is required to interact with the client mobile application and the public API service. The blockchain, once set up, has not been tested if it works without internet connectivity, but setting up the blockchain requires internet connection every time.

A Linux-based OS, preferably either Ubuntu or Xubuntu, is required for running the blockchain locally.

V. Results

This section is split into three main parts: (i) system architecture overview, (ii) client mobile app functionalities, and the (iii) public API service.

A. System Architecture Overview

The system consists of 5 primary components: (i) blockchain network, (ii) client mobile application, (iii) main server, (iv) secondary server, and the (v) off-chain slave database.

A.1 Blockchain Network

```
(base) Numbers06@baseubuntu20:~/Personal/GitHub/sp-porkwatch$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                                NAMES
d841deea441c   registry:2                          "/entrypoint.sh /etc..." 3 hours ago   Up 3 hours   127.0.0.1:5000->5000/tcp           kind-registry
0e5285c9f57b   kindest/node:v1.24.4                "/usr/local/bin/entr..." 3 hours ago   Up 3 hours   0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp, 127.0.0.1:8888->6443/tcp   kind-control-plane
```

Figure 10: Docker containers for managing the kubernetes cluster and related images

NAME	IP	NODE	AGE
cc-ong1msp-ong1-peer1porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.54	kind-control-plane	3h2m
cc-ong1msp-ong1-peer1porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.61	kind-control-plane	150m
cc-ong1msp-ong1-peer2porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.60	kind-control-plane	3h2m
cc-ong1msp-ong1-peer2porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.65	kind-control-plane	150m
cc-ong2msp-ong2-peer1porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.53	kind-control-plane	3h2m
cc-ong2msp-ong2-peer1porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.68	kind-control-plane	150m
cc-ong2msp-ong2-peer2porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.55	kind-control-plane	3h2m
cc-ong2msp-ong2-peer2porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.67	kind-control-plane	150m
cc-ong3msp-ong3-peer1porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.57	kind-control-plane	3h2m
cc-ong3msp-ong3-peer1porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.63	kind-control-plane	150m
cc-ong3msp-ong3-peer2porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.59	kind-control-plane	3h2m
cc-ong3msp-ong3-peer2porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.64	kind-control-plane	150m
cc-ong4msp-ong4-peer1porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.56	kind-control-plane	3h2m
cc-ong4msp-ong4-peer1porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.66	kind-control-plane	150m
cc-ong4msp-ong4-peer2porkwatch-27fc165b33f888c782187cad6c7c4570	10.244.0.58	kind-control-plane	3h2m
cc-ong4msp-ong4-peer2porkwatch-499bc9f49b685ea23bd67e09e936535f	10.244.0.62	kind-control-plane	150m
fabric-operator-78bc4d9495-r9nj5	10.244.0.10	kind-control-plane	3h10m
hlfc-console-84d4f4495-tsx5d	10.244.0.52	kind-control-plane	3h5m
ong0-ca-649947976-ghkox	10.244.0.12	kind-control-plane	3h10m
ong0-orderernode1-7c7bfc97d-dmf81	10.244.0.28	kind-control-plane	3h9m
ong0-orderernode2-58cc88d568-klh81	10.244.0.35	kind-control-plane	3h9m
ong0-orderernode3-7846b6d966-rgrp9	10.244.0.39	kind-control-plane	3h9m
ong1-ca-8676c7648-5xh39	10.244.0.15	kind-control-plane	3h10m
ong1-peer1-7c656887bb-wesxd	10.244.0.25	kind-control-plane	3h9m
ong1-peer2-6df572885b-2b5d	10.244.0.32	kind-control-plane	3h9m
ong2-ca-76b9f8998-1h1kd	10.244.0.17	kind-control-plane	3h10m
ong2-peer1-7b7454f97-12r9h	10.244.0.37	kind-control-plane	3h9m
ong2-peer2-77f9b88878-j6n48	10.244.0.42	kind-control-plane	3h8m
ong3-ca-54684589b-kf2dd	10.244.0.19	kind-control-plane	3h10m
ong3-peer1-5c46b8576d-2kd7v	10.244.0.45	kind-control-plane	3h8m
ong3-peer2-5bbf449f8-6l4hz	10.244.0.48	kind-control-plane	3h8m
ong4-ca-849df45f76-n6ftt	10.244.0.20	kind-control-plane	3h10m
ong4-peer1-7f8cb8d88b-xozq7	10.244.0.50	kind-control-plane	3h8m
ong4-peer2-78b86c7958-88h4j	10.244.0.49	kind-control-plane	3h8m

Figure 11: K9s terminal based UI displaying the pods for the blockchain network

The blockchain aspect of the system utilizes Hyperledger Fabric’s modular architecture to deploy peer nodes, orderer nodes, certificate authorities, and other such components. Shown in Fig. 11 above are the pods maintaining the blockchain network, they are as follows:

1. **cc-org***: Each pod runs a version of the chaincode for each of the two peers in each of the four peer organizations.
2. **fabric-operator-***: Manages and operates the Hyperledger Fabric blockchain network within the Kubernetes cluster.
3. **hlf-console-***: Provides a web-based interface for users to interact with, monitor, and perform administrative tasks over the Hyperledger Fabric network.
4. **org*-ca***: Runs the certificate authority for each of the four peer organizations and one orderer organization.
5. **org0-orderersnode***: Runs each of the three orderer nodes managed by the orderer organization 0, which handles ordering of transactions into blocks and establishing consensus deterministically within the network, using the crash fault tolerant **Raft** consensus algorithm. The 3 orderer nodes together form the ordering service for the channel **C1**.
6. **org*-peer***: Runs each of the two peer nodes in each of the four peer organizations; peer nodes maintain a copy of the distributed ledger, execute, endorse and validate transactions, and participate in the consensus process.

All 8 peer nodes execute transactions using a single chaincode, containing all 10 smart contracts in the network, and serve to allow the main server and the mobile client application to perform read and write operations to the blockchain.

In the network, all nodes, consisting of the orderer and peer nodes, are joined to a single unified channel. This channel serves as a communication pathway for

coordinating and processing transaction proposals, which are separately verified by the peer nodes, and processed into blocks by the orderer nodes, before committing to each of the distributed ledger instances on all of the peer nodes. These processes serve to ensure that the codebase has not been tampered with, that each transaction proposal is valid, and that data is consistent across the blockchain network.

Additionally, the blockchain network makes use of the document-oriented NoSQL CouchDB as its state database. The state database occupies a part of the distributed ledger alongside the blockchain, and a copy is present and updated in all peers for every update to the ledger. Unlike the default LevelDB database, CouchDB allows for rich queries and modelling of data in JSON format, both of which are necessary to the client mobile application.

```

2023-06-26T07:14:02.529Z info [c-api:lib/handler.js] [mychannel-902460c3] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.user.GetUser
isTransactionSubmitting false
matches[1] arcilla@example.com
clientCommonName arcilla@example.com
2023-06-26T07:14:05.049Z info [c-api:lib/handler.js] [mychannel-79d73f4c] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.location.getLocation
isTransactionSubmitting false
matches[1] arcilla@example.com
clientCommonName arcilla@example.com
2023-06-26T07:14:05.069Z info [c-api:lib/handler.js] [mychannel-45cfff1e4] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.user.GetUser
isTransactionSubmitting false
matches[1] arcilla@example.com
clientCommonName arcilla@example.com
transactionName org.porkwatch.user.GetUser
isTransactionSubmitting false
matches[1] arcilla@example.com
clientCommonName arcilla@example.com
2023-06-26T07:16:07.505Z info [c-api:lib/handler.js] [mychannel-a5502b87] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
2023-06-26T07:16:07.506Z info [c-api:lib/handler.js] [mychannel-0778039a] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.user.GetUser
isTransactionSubmitting false
matches[1] arcilla@example.com
clientCommonName arcilla@example.com
transactionName org.porkwatch.user.GetUser
isTransactionSubmitting false
matches[1] arcilla@example.com
clientCommonName arcilla@example.com
2023-06-26T07:16:07.522Z info [c-api:lib/handler.js] [mychannel-d80b7c1b] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
2023-06-26T07:16:07.522Z info [c-api:lib/handler.js] [mychannel-d3132403] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.buyorder.CreateBuyOrderWithStringState
isTransactionSubmitting true
matches[1] emmataylor@example.com
clientCommonName emmataylor@example.com
matches[1] emmataylor@example.com
2023-06-26T07:23:59.369Z info [c-api:lib/handler.js] [mychannel-69a15284] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.transfer.CreateTransferWithStringState
isTransactionSubmitting true
matches[1] emmataylor@example.com
clientCommonName emmataylor@example.com
matches[1] emmataylor@example.com
2023-06-26T07:24:02.298Z info [c-api:lib/handler.js] [mychannel-97f51752] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
transactionName org.porkwatch.transfer.UpdateTransferWithStringState
isTransactionSubmitting true
matches[1] emmataylor@example.com
clientCommonName emmataylor@example.com
matches[1] emmataylor@example.com
2023-06-26T07:29:51.038Z info [c-api:lib/handler.js] [mychannel-f528bc66] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer
(base) Numbers0@Basetuntu20:~/Personal/GitHub/sp-porkwatch/blockchain$

```

Figure 12: Transactions in the Blockchain (with custom console.log prints every invocation)

A..2 Client Mobile Application

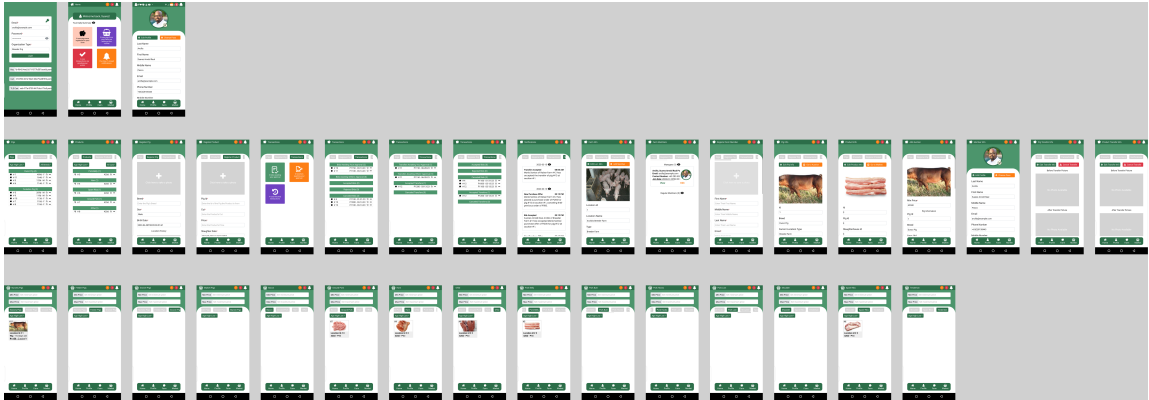


Figure 13: PorkWatch Mobile App Screens

The PorkWatch mobile app is built using the React Native Expo framework and acts as an interface for supply chain actors to interact with the blockchain. Its functionalities include managing user profiles, managing their pigs, products, location, participating in auctions, and more. As any interaction with the blockchain requires authentication, the mobile app allows only logged-in users to access its screens, with the login screen serving as the entry point.

In all, the mobile application hosts 35 screens, the visibility and functionalities of which depends on the user visiting the screen, e.g., the "Register Pig" screen for breeders, the screens for viewing products for butchers and retailers, and so on.

1. **Login** - Asks for the user's email, password, organization type, key, certificate, and the TLS certificate of their organization for authentication.
2. **Home** - Where users are redirected to after a successful authentication.
3. **Profile** - Where users can view and edit their own profile details.
4. **Farm** - Groups together screens relevant to the managing of pigs, products, users, locations, and all things related, e.g., bids, transactions, etc.

- (a) **Pigs** - Where users can view and manage their location's pigs.
- (b) **Products** - Where users can view and manage their location's products.
- (c) **Register Pig** - Where breeders can register a new pig into the system.
- (d) **Register Product** - Where butchers can register a new product into the system.
- (e) **Transactions** - Where users can view their ongoing and past transactions, that is, the bids on their pig auctions, the transfers for their pigs and/or products, and so on.
- (f) **Notifications** - Where users can view their notifications.
- (g) **Farm Info** - Where users can view and modify, should they be managers, their location's information.
- (h) **Farm Members** - Where users can view and manage, should they be managers, their location's members.
- (i) **Register Farm Member** - Where managers can register new members into their location.
- (j) **Pig Info** - Where users can view and edit, should their location own the pig, a pig's detailed information, and access the pig's historical information across the supply chain.
- (k) **Product Info** - Where users can view and edit, should their location own the product, a product's detailed information, and access the product's historical information across the supply chain.
- (l) **Add Auction** - Where breeders and raisers can post a new auction for their location's pigs.
- (m) **Auction Info** - Where breeders, raisers, and butchers can view a pig auction's details, place or manage bids, and so on.

- (n) **Member Info** - Where users can view and edit, should they be managers of the same location as the user, another user's profile.
 - (o) **Pig Transfer Info** - Where breeders, raisers, and butchers can view and approve/cancel a pig transfer from or into their location.
 - (p) **Product Transfer Info** - Where butchers and retailers can view and approve/cancel a product transfer from or into their location.
5. **Market** - Groups together screens displaying pigs and products that are currently for sale.
- (a) **(4) Pig Screens** - Where breeders, raisers, and butchers can view currently active pig auctions categorized by weight classification (e.g., nursery pigs, feeder pigs).
 - (b) **(11) Product Screens** - Where butchers and retailers can view the products currently being sold categorized by cut type (e.g., pork chops, spare ribs).

A..3 Main Server

The main server acts as the intermediary between the blockchain network and the client mobile app, facilitating communication and interaction between the two. Responsibilities of the main server include:

1. **Authenticating and authorizing clients** - The main server ensures that the identity and credentials of the mobile app users are valid by verifying their key, certificate, and TLS certificate. Once authenticated, the main server issues a signed JSON Web Token (JWT) to the clients, which serves as proof of their identity, and appended to their every request as part of the request header. This token allows them to interact with the blockchain for a full hour, before they are required to authenticate again.

2. **Handling client requests** - The main server receives requests from the client mobile app, such as submitting transactions, querying blockchain data, and initiating smart contract invocations.
3. **Interacting with the blockchain network** - The token given to the user upon successful authentication contains data that would let the server instantiate a new connection to the blockchain through a certain gateway peer depending on the user's organization type before any transaction with the blockchain proceeds, and close said connection prior to sending the response back to the client mobile app.
4. **Transaction processing and validation** - Upon receiving a transaction request or a query from the client mobile app, the main server performs necessary validation checks and business logic processing, ensuring that the transaction data adheres to the defined rules of the blockchain network and the asset models the targeted smart contracts make use of before submitting it to the blockchain network for consensus and inclusion in the distributed ledger.
5. **Event listening** - The main server listens for events emitted within the smart contracts of the blockchain network, processes their payload, and handles insertion to the off-chain slave CouchDB database to keeping it up-to-date with the on-chain state databases.

By acting as the intermediary, the main server simplifies the interaction between the Hyperledger Fabric blockchain network and the client mobile app, abstracting the complexity of the underlying blockchain technology and providing a seamless user experience.

```

blockNumber: 43n,
chaincodeName: 'porkwatch',
eventName: 'CreateTransfer',
transactionId: '97f51752a1b1ff9310b997963937a737d841d81bce3b541f7dd0f19e6f695d1',
payload: '{"AcceptedByBuyerId":"","AcceptedBySellerId":"","AcceptedDate":"","AfterTransferPicture":"","AuctionId":"","BeforeTransferPicture":"","BidId":"","BuyOrderId":"2","BuyerId":"4","CanceledByBuyerId":"","CanceledBySellerId":"","Id":"2","PigId":"4","Price":100,"ProductIds":["9"],"SellerId":"3","StartDate":"2023-06-26T07:24:02.174Z","TransferDate":"","TransferFromId":"3","TransferToId":"4","docType":"transfer","key":"\\u0000transfer\\u00002\\u0000"}'
}
Document with _id 'transfer2' already exists. Skipping creation.
{
  blockNumber: 44n,
  chaincodeName: 'porkwatch',
  eventName: 'UpdateTransfer',
  transactionId: 'bac33d73979b7ea907a9661a0fc38120f1b7bf2011b31f47d6caf958313fa364',
  payload: '{"AcceptedByBuyerId":"","AcceptedBySellerId":"3","AcceptedDate":"","AfterTransferPicture":"http://20.89.188.190:3001/public/uploads/2023-06-26T07-26-54.230ZJohn's Slaughterhouse-3-2-Transfer-BeforeTransferPicture.jpg","BidId":"","BuyOrderId":"2","BuyerId":"4","CanceledByBuyerId":"","CanceledBySellerId":"","Id":"2","PigId":"4","Price":100,"ProductIds":["9"],"SellerId":"3","StartDate":"2023-06-26T07:24:02.174Z","TransferDate":"","TransferFromId":"3","TransferToId":"4","docType":"transfer","key":"\\u0000transfer\\u00002\\u0000"}'
}
{
  blockNumber: 45n,
  chaincodeName: 'porkwatch',
  eventName: 'UpdateTransfer',
  transactionId: 'f528bc660f3b66f02fd6a1278ed78ba02574cf0c3305a893507f0333287a0f7',
  payload: '{"AcceptedByBuyerId":"4","AcceptedBySellerId":"3","AcceptedDate":"","AfterTransferPicture":"http://20.89.188.190:3001/public/uploads/2023-06-26T07-29-50.371ZEmma's Retail Stone-4-2-Transfer-AfterTransferPicture.jpg","AuctionId":"","BeforeTransferPicture":"http://20.89.188.190:3001/public/uploads/2023-06-26T07-26-54.230ZJohn's Slaughterhouse-3-2-Transfer-BeforeTransferPicture.jpg","BidId":"","BuyOrderId":"2","BuyerId":"4","CanceledByBuyerId":"","CanceledBySellerId":"","Id":"2","PigId":"4","Price":100,"ProductIds":["9"],"SellerId":"3","StartDate":"2023-06-26T07:24:02.174Z","TransferDate":"","TransferFromId":"3","TransferToId":"4","docType":"transfer","key":"\\u0000transfer\\u00002\\u0000"}'
}
{
  blockNumber: 46n,
  chaincodeName: 'porkwatch',
  eventName: 'UpdateProduct',
  transactionId: '16347a28e6ed14e210ea0ee282735679175ace3acbd56934756f7526c74fab3',
  payload: '{"Cut":"Bacon","Id":"9","Picture":"http://20.89.188.190:3001/public/uploads/2023-06-26T07-21-00.233ZJohn's Slaughterhouse-3-Product-Picture.jpg","PigId":"4","Price":100,"RegisteredById":"3","RegistrationDate":"2023-06-26T07:21:00.955Z","RetailerId":"4","RetailerLocationId":"4","SlaughterhouseId":"3","TransferDate":"","docType":"product","key":"\\u0000product\\u00009\\u0000"}'
}
}

```

Figure 14: Events Emitted by the Smart Contracts in the Blockchain Network

A..4 Secondary Server

The secondary server serves as a public API service accessible to all users, including the public and other stakeholders. It facilitates comprehensive queries within the system, allowing other applications, particularly dashboards, to connect and perform data analysis on the supply chain managed by the system. By utilizing an off-chain slave database, this server enables faster and more convenient access to data. However, it solely permits queries to the off-chain slave database and does not support any write operations.

A..5 Off-chain Slave Database

Managed by the main server, the off-chain slave CouchDB database remains synchronized with the latest data from the on-chain state databases in real-time. This synchronization ensures that the database is constantly updated. The secondary server leverages this off-chain slave database to provide real-time access to the system's public API service, delivering up-to-date information to users.

B. Client Mobile App Functionalities

This section shows the different functionalities of the client mobile app PorkWatch, first showing how authentication works, given that authentication is necessary for any operation involving the blockchain, with the next parts following the order as written in the use cases section.

B.1.1 Authentication

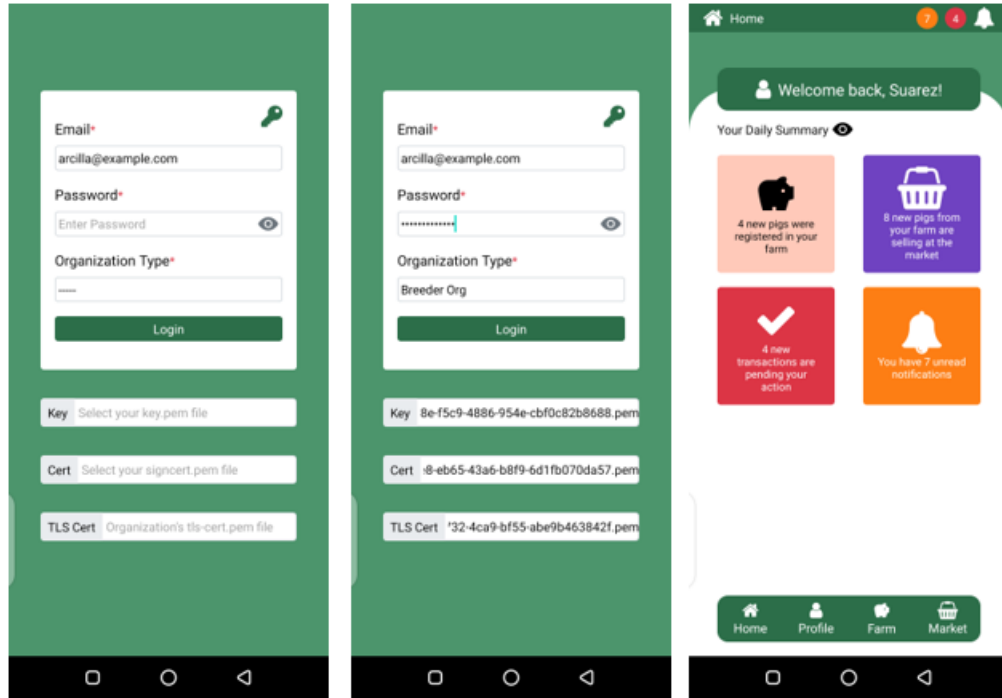


Figure 15: Logging in

Only users with both login credentials and their respective key, certificate, and their organization's TLS certificate can log into the mobile and instantiate a connection to the blockchain network. Axios sends all 6 fields to the main server at once, where the server would do as follows:

1. Create files from the received key, certificate, and TLS certificate, each with a different string generated from the collision resistant uuid v4 as filename.
2. If the connection to the blockchain network is successful through said 3 files, i.e., the stated organization's certificate authority has verified the sent X.509 certificate, proceed with querying the ledger for a user with a matching email.
3. If a user with a matching email is found, query the ledger for a still active password with a matching user id.

4. If a password with a matching user id is found, use the bcrypt library in the server to check if the password sent from the mobile app and the password in the ledger, which is encrypted using 10 salt rounds, matches.
5. If the request passes all of the above checks, an access token is generated signed using the JWT library, containing the user's email, id, organization type, the paths to their newly created key, certificate, and TLS certificate files, using a JWT secret, which is stored in an untracked env file. The access token is given an expiration of an hour, after which the user will have to redo the authentication process.
6. The access token and the user's details, at the time of query, are sent as response to the mobile app, where they will be managed and made available by redux and persistently stored in the device's disk using the react native async storage library.
7. The user, after receiving their access token and user details, are then redirected to the home screen.

Following this, any subsequent interactions with the blockchain network, which happens in every screen in the mobile app, will make use of the 3 uuid v4 generated strings stored in the user's access token to connect to the blockchain network, while making use of the user id in the same access token to authenticate in the main server, i.e., authentication happens on both the main server and blockchain levels.

This protects the system from bruteforce attacks to find a valid email and password combination, as the attacker will have to first find a valid combination of key, certificate, and TLS certificate, all the while being unable to connect to the blockchain network. Aside from a valid combination of the three files, authentication will also not work if the 3 files and the email, if ever the valid password counterpart was found, do not match. It is mentioned earlier in the blockchain section that all smart contracts

first process the transaction request and match the user credentials and the X.509 certificate used to connect to the blockchain network.

B.2 Adding a New Pig to the User's Location

Depending on the type of the location the user is registered under, there are two ways to add a new pig to the user's location: (i) by registering them, or by (ii) getting a successful transfer. This part shows the former, however, only users with the role of breeders are allowed to do this part, that is, registering pigs into the system.

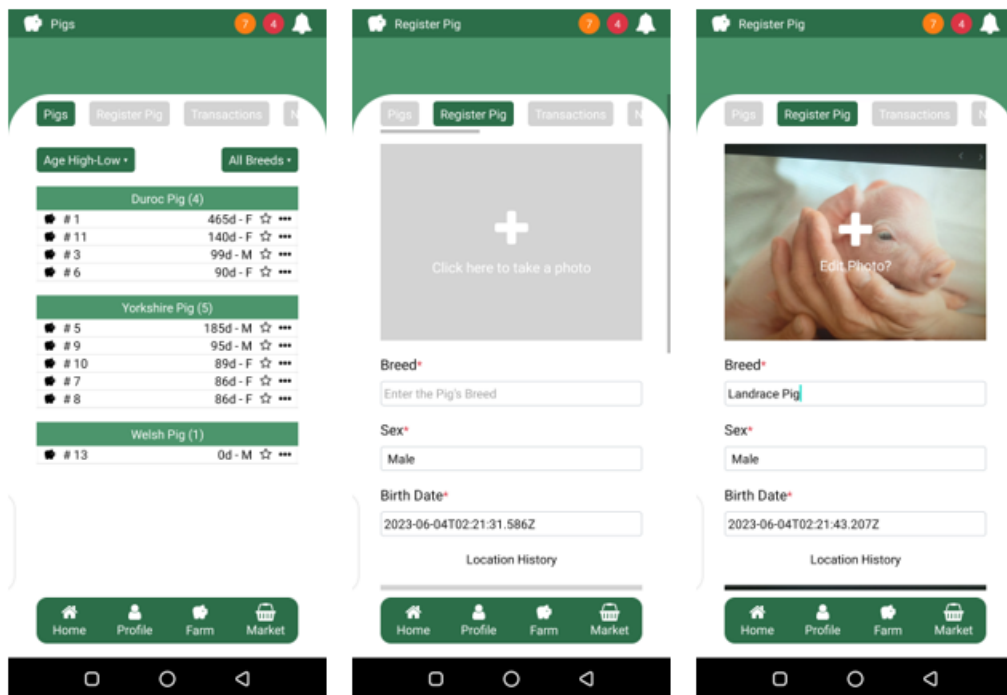


Figure 16: Adding a new pig

To register a new pig, the breeder will first navigate to the "Register Pig" screen. There, they will fill the necessary fields, and press on the empty image slot, which will open up a camera view, to take of a picture of the new pig.

Once all of that has been accomplished, the user is to press on the "Register Pig?" button at the very bottom. If the registration was successful, the new pig will appear

in the "Pigs" screen, automatically sorted into its own group by breed. There, the user can press on the ellipsis icon on the row's rightmost part, where they will be presented the option to view the pig, or in the case of a manager, they will be shown both the view and edit options.

B.3 Adding a New Product to the User's Location

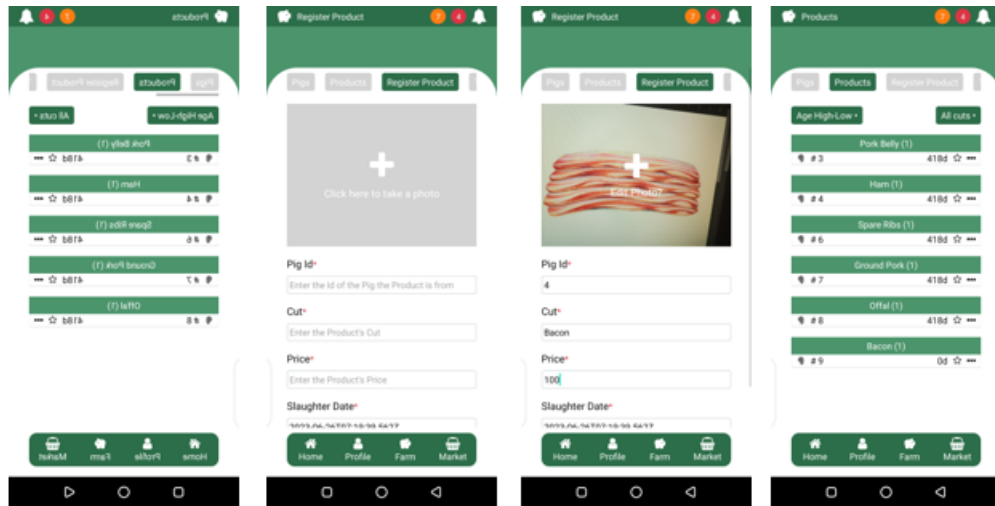


Figure 17: Adding a new product

In the same way as in registering a new pig, only butchers can register new products into the system. Retailers can only add new products into their location after a successful product transfer from a slaughterhouse. This part shows the former, through the user Smith who is a butcher.

To register a new product, the butcher will first navigate to the "Register Product" screen. The rest of the proceeds as in registering a new pig in the case of breeders, where the butcher fills the form and takes a picture of the product, and the product would, having been successfully registered, appear in the "Products" screen of the user. There, the user can also press on the ellipsis icon on the row to show options for navigating to the screen for viewing and editing the newly registered product's

details.

B.4 Updating a Pig

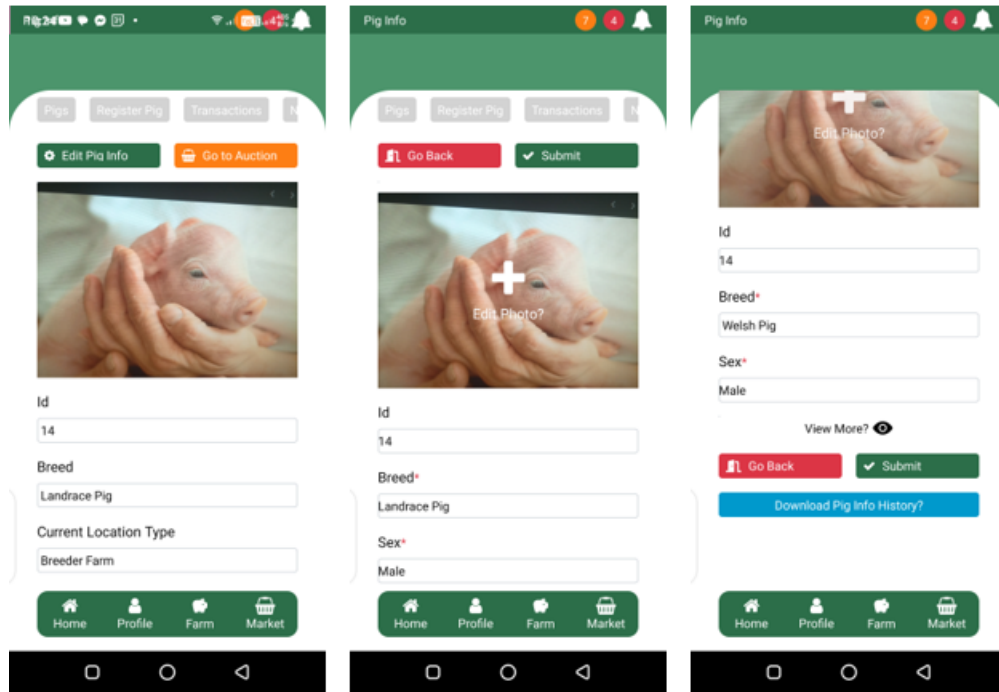


Figure 18: Updating a pig

To update a pig, the user will first have to navigate to the "Pig Info" screen. Here, the user navigated to said screen by pressing on the "View Pig Info" option from the menu that popped up after the ellipsis icon of the relevant row is pressed in the "Pigs" screen.

Now in the "View Pig Info" screen, the user can press on "Edit Pig Info" to switch to edit mode. There, once all required fields are filled in and yup does not detect any errors, the user can then press on the "Submit" button. If the main server and blockchain do not detect errors, the pig will successfully be modified and the mode will be switched back to view mode, with the updated pig details showing on the fields.

B..5 Viewing Pig History

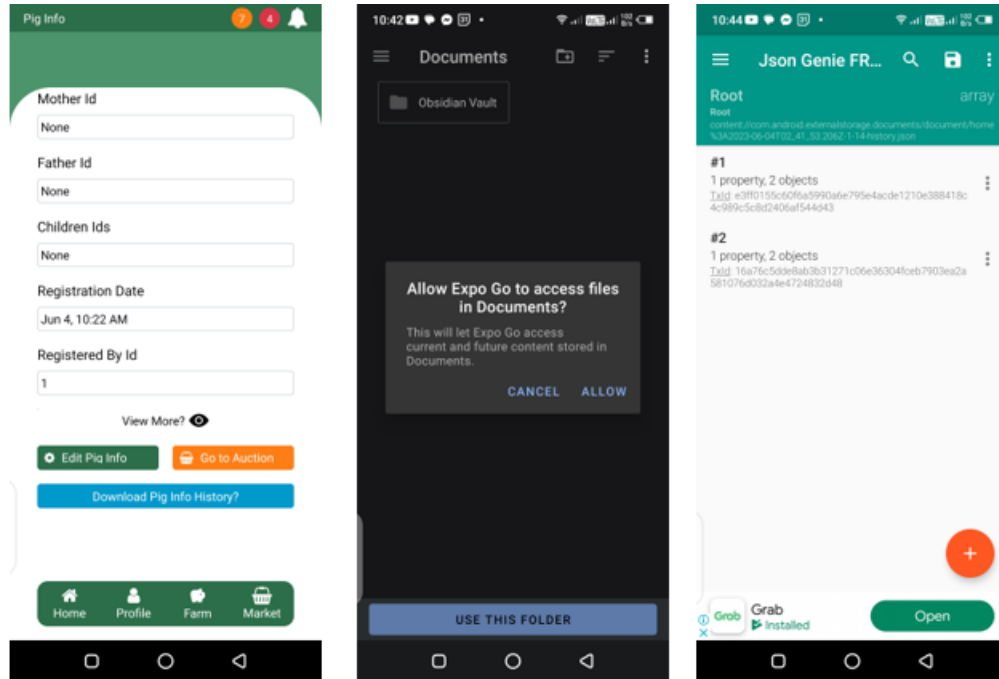


Figure 19: Viewing Pig History

To view a pig's history, navigating to the relevant "Pig Info" comes first. Once there, regardless if it's the view mode or the edit mode, pressing the "Download Pig Info History?" button will download the file in your directory of choice, after taking your permission.

As the downloaded file is in JSON format, a JSON viewer, like JSON Genie in this case, is necessary to view the file. Fig. 19 shows the two versions so far of the selected pig, alongside the timestamp indicating when the update took place. The history file itself is in an array format, with the latest version of the pig at the top, and the oldest version at the bottom.

B..6 Viewing Product History

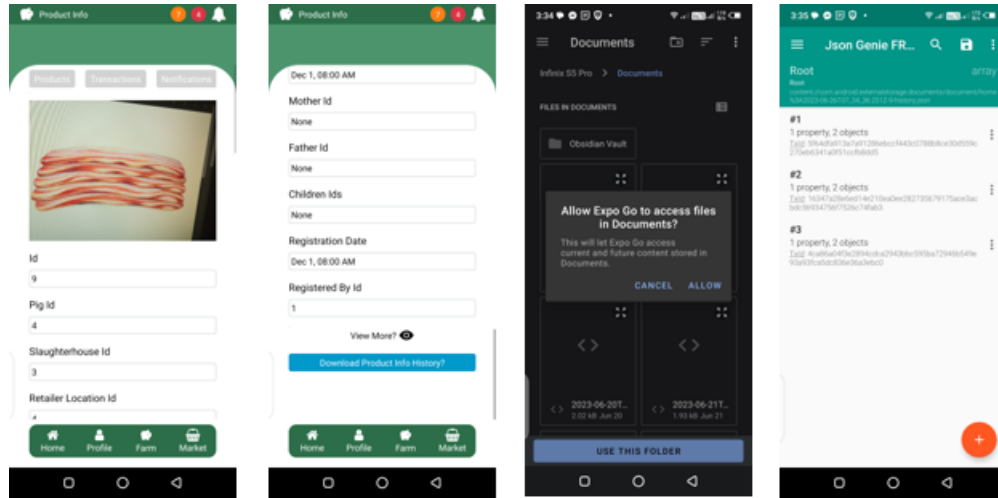


Figure 20: Viewing Product History

Viewing a product's history also has the same procedure as in viewing a pig's history, but instead of navigating to a "Pig Info", the user is to navigate to a "Product Info" and press the "Download Product Info History?" at the bottom of the screen. The downloaded file is also in JSON format, and includes the history of both the product asset itself and the pig asset it is from, that is, the source of the product. This enables whole-of-chain traceability in the system, as the product can be traced back to the farm it is from.

B..7 Adding a Pig Auction

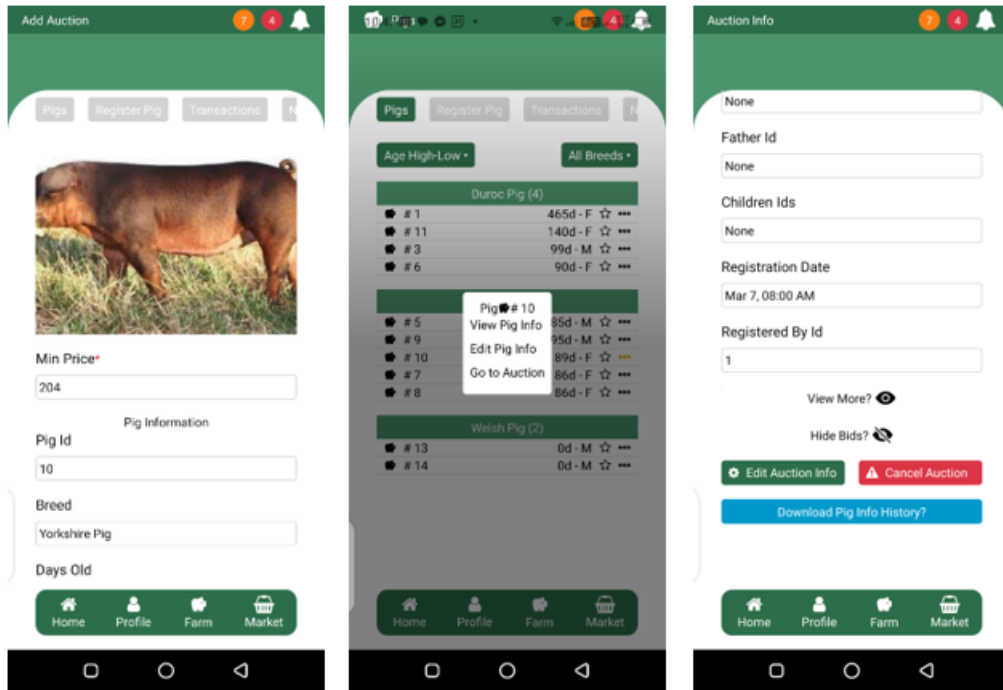


Figure 21: Adding a Pig Auction

To add a pig auction, first, the user has to navigate to either the "Pigs" or "Pig Info" screen. In this case, the user navigated to the "Pigs" screen. There, pressing on the rightmost ellipsis icon of the relevant row will show a menu modal. If the pig has neither an active auction nor an active transfer, the menu will include a "Add auction" option. Pressing on this will redirect the user to the "Add Auction" screen.

The form on the "Add Auction" screen only requires setting the minimum price for the auction. The price is automatically calculated from the latest measured weight of the pig by multiplying it for P170 per kilogram. This is editable however, and once the user has pressed on the "Create Auction?" button, the user will be redirected to the "Auction Info" screen, populated with the details of the newly created pig auction.

Once there, the user can view the bids for the auction, and approve or reject them

as necessary. The user is also able to edit the auction details, i.e., edit the minimum price, which automatically sets the minimum price of the bids to be created by users from other locations. Cancelling the auction is also an option, and if there were active bids at the time of cancelling, said bids will automatically appear as rejected on the members of the locations that created them.

B.8 Bidding in a Pig Auction

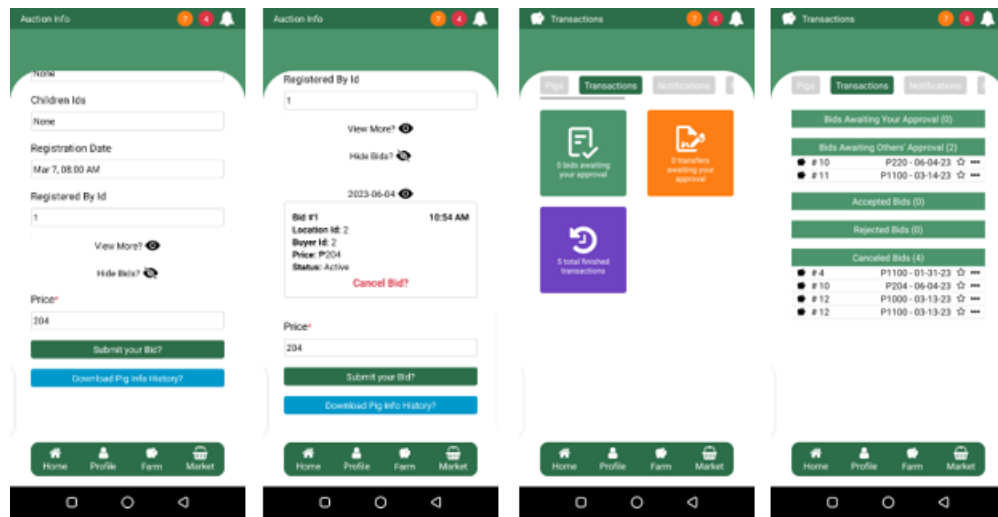


Figure 22: Bidding in a Pig Auction

Users from locations other than the initiator of the pig auction can bid for the pig auction by first navigating to the appropriate screen under the "Market" tab. Based on weight, for example, given that pig id # 10, the pig in the example, only has a latest measured weight of 1.2kg, its auction can be found in the "Nursery Pigs" screen. There, the user is to press on the picture of the auction to navigate to its corresponding "Auction Info" screen.

Navigating to the very bottom of the screen, past bids on the auction are visible. As there are none at the time, no bids are displayed. The user can submit their own bid through the form at the bottom, where they only need to place their price in

order to submit a new bid. The yup validator schema for the form will check if the submitted price is either equal or higher than the current price of the pig auction, which is automatically set based on the highest priced bid so far. If the new bid passes this check, the bid will successfully be added to the auction's bids and will appear on the screen.

The user can choose to submit a new bid, after the successful submission of which the user's location's older bids will automatically be cancelled. The user can then check for their recently created bid and their location's other bids through the "Transactions" screen. Its bids mode will display all of the user's location's bids and to which pig and auction they are for.

B.9 Accepting/Rejecting a Bid

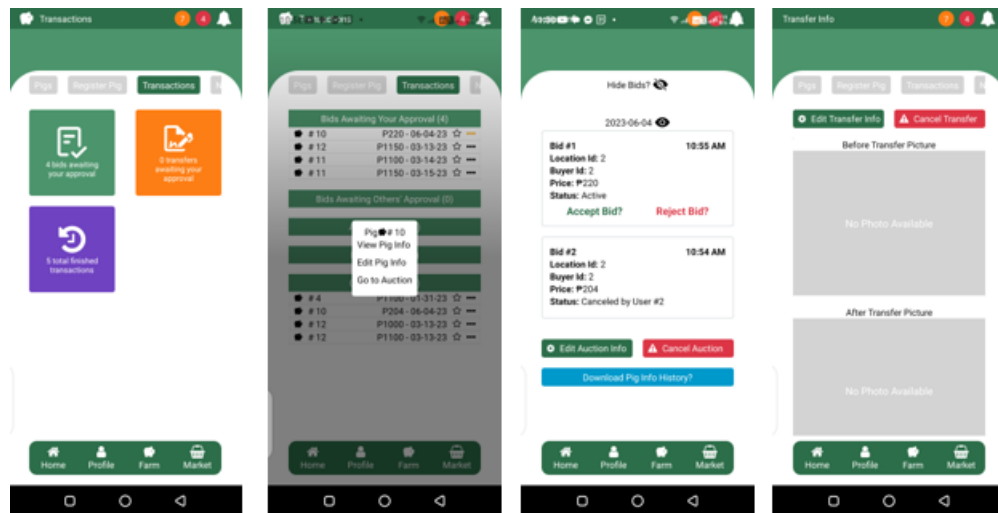


Figure 23: Accepting a Bid

Users from the location that initiated the pig auction are able to accept a bid through the "Auction Info" screen. Here, the user navigated to said screen through the "Transactions" screen, where the initial mode shows that the user has bids pending their approval. Upon pressing this, the location's bids are displayed.

The bid in question is for pig id # 10. Pressing the ellipsis icon to the rightmost part of the row and pressing the "Go to Auction" option will redirect the user to the appropriate "Auction Info" screen. There, the user will navigate to the very bottom of the screen and accept the bid of their choice.

Once a bid is accepted, the user is automatically redirected to a "Transfer Info" screen populated with details of the newly created transfer state. While it is not shown here, the bid will have been removed from the pending bids list of Suarez and his fellow members, and moved to the accepted bids table. The same goes for the members of the location that submitted the bid.

Should the user decide to have instead rejected the bid. At the bottom part of the "Auction Info" screen, the user could have pressed on the "Reject Bid?" link instead. This will have updated the bid to a rejected state, and will promptly appear as "Rejected by User #1" in Suarez' screen.

B.10 Confirming a Pig Transfer

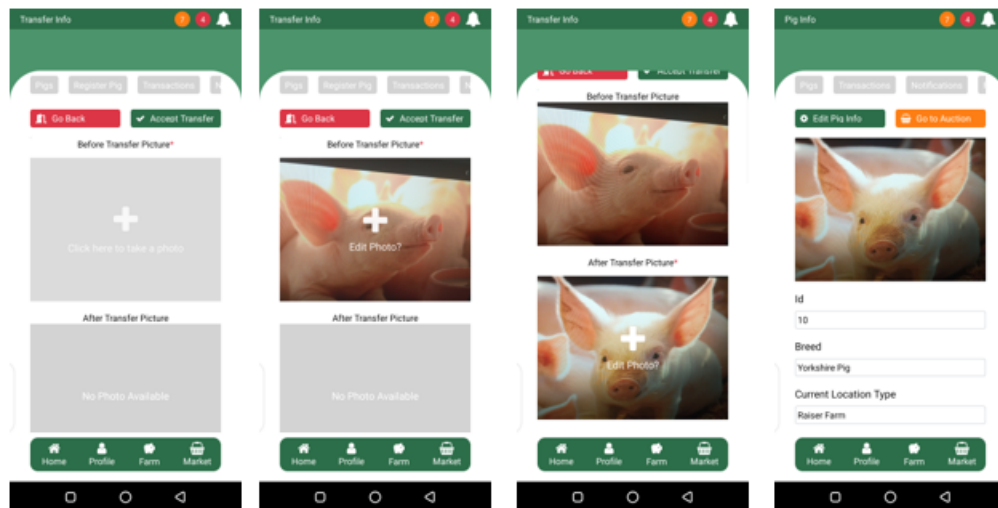


Figure 24: Confirming a Pig Transfer

Users from both the location that initiated the pig auction and the location that submitted the accepted bid should confirm the resulting transfer after the bid is accepted for the pig transfer to take effect, that is, for the pig to be removed from the "Pigs" screen of the members of the original location, and for the pig to appear in the "Pigs" screen of the members of the location that submitted the bid.

For this, the user navigated to the "Transfer Info" screen through the "Transactions" screen. The initial mode of the latter screen includes a button to switch to the transfers mode and make the transfer visible.

Once in the appropriate transfers mode, the user can navigate to the transfer in question by pressing on the rightmost ellipsis icon on the row. The user then selects the "Go to Transfer" option in the ensuing modal and is redirected to the appropriate "Transfer Info" screen afterwards.

Now in the "Transfers Info" screen, as the user, Suarez, is from the location that initiated the auction, they are to press on "Edit Transfer Info" to continue. Once pressed, the picture slot becomes pressable, the "Before Transfer Picture" slot in Saurez' case, indicating the picture of the pig before the transfer has commenced.

Once the picture slot is pressed, the camera is opened, and a picture of the pig is to be taken. Once taken, the user can already press on "Accept Transfer", as the picture is the sole requirement of confirmation. The screen is then toggled back to view mode.

Navigating back to the "Transactions" screen, the user can see that the transfers pending their approval have been reduced by one, and the transfers mode reveals that the transfer now only requires the approval of the members from the other location.

As transfers require confirmation from both sides, a user from the other location, Maria, is to log in. They then navigate to the "Transactions" screen and into transfers mode, pressing on the same buttons to navigate to the "Transfer Info" screen.

There, they press on "Edit Transfer Info" and press the touchable "After Transfer

Picture” slot. This opens the camera view, and after they capture a picture of the pig after the transfer has been done in actuality, they can then press on ”Accept Transfer”. This whole process indicates that the transfer has gained the tacit approval of both locations, and that the digital confirmation of the transfer signals the actual transfer of the pig in the real world.

While not shown here, the pig has then been removed from the ”Pigs” screen of the original location, and has appeared in the same screen of the members of the location that submitted the bid.

B.11 Confirming a Product Transfer

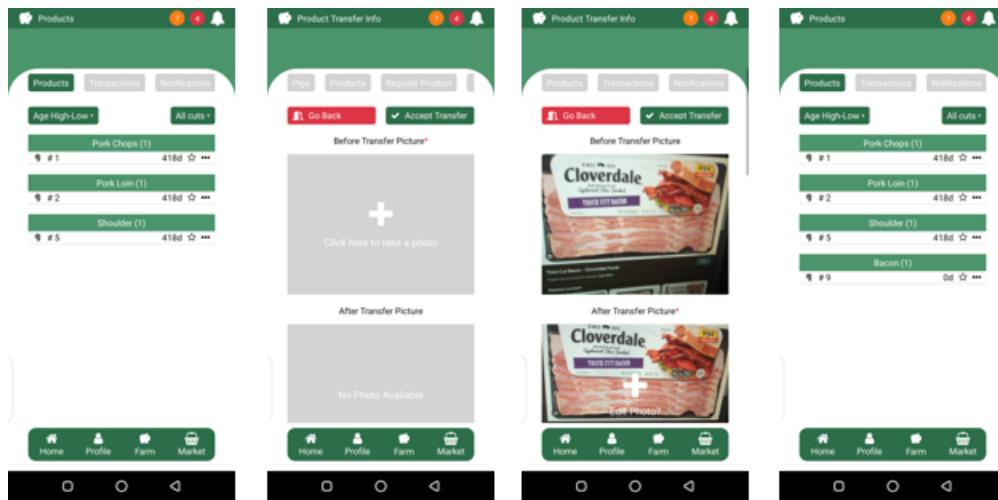


Figure 25: Confirming a Product Transfer

The process for confirming a product transfer also requires the approval of both a user from the participating slaughterhouse and retailer locations, the entire process of which is also similar to confirming a pig transfer. The product, after a successful transfer, also disappears from the ”Products” section of the users in the slaughterhouse, meanwhile appearing in the ”Products” section of the users in the retailer location.

B..12 Registering a New User

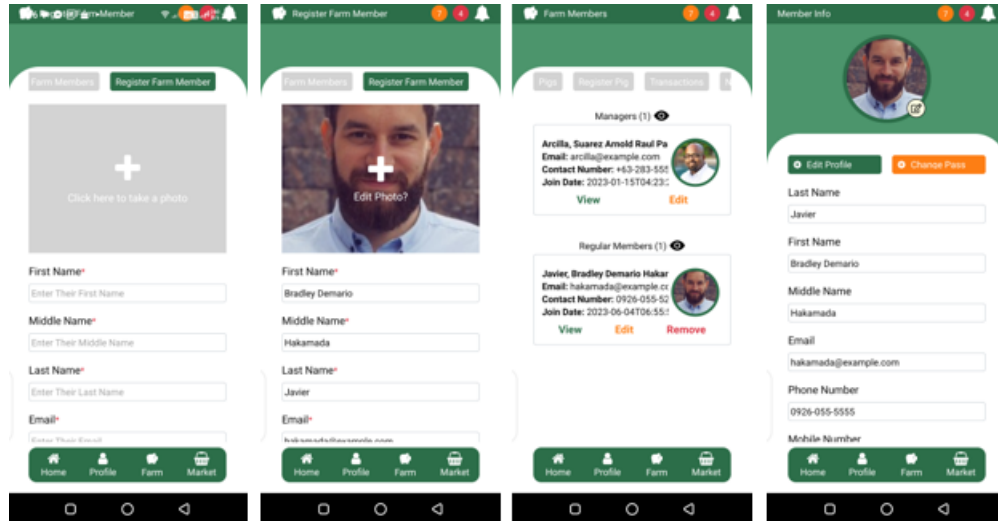


Figure 26: Registering a New User pt.1

Only location managers can register new users into their location. For this, the user is to navigate to the "Register Farm Member" screen. There, they are to fill the required fields and select a picture of the new user from their phone's filesystem. Once all required fields have been accomplished according to the yup validator schema, the user can press on "Register Farm Member?". If neither the main server nor the blockchain detect any issue, the user and their password will be created, and the user's location will be modified to include the id of the newly added user into the location.

```
(base) Numbers00@BaseUbuntu20:~/Personal/GitHub/sp-porkwatch/blockchain$ chaincodes/porkwatch/scripts/general/enroll_user.sh 1 hakamada@example.com WatchPork123!
2023/06/04 11:59:22 [INFO] Configuration file location: /home/Numbers00/.fabric-ca-client/fabric-ca-client-config.yaml
2023/06/04 11:59:22 [INFO] TLS Enabled
2023/06/04 11:59:22 [INFO] TLS Enabled
Error: Response from server: Error Code: 74 - Identity 'arcilla@example.com' is already registered

2023/06/04 11:59:23 [INFO] TLS Enabled
2023/06/04 11:59:23 [INFO] generating key: &{A:ecdsa S:256}
2023/06/04 11:59:23 [INFO] encoded CSR
2023/06/04 11:59:24 [INFO] Stored client certificate at /home/Numbers00/Personal/GitHub/sp-porkwatch/blockchain/infrastructure/sample-network/temp/enrollments/org1/users/arcilla@example.com/msp/signcerts/cert.pem
2023/06/04 11:59:24 [INFO] Stored root CA certificate at /home/Numbers00/Personal/GitHub/sp-porkwatch/blockchain/infrastructure/sample-network/temp/enrollments/org1/users/arcilla@example.com/msp/cacerts/test-network-org1-ca-ca-localho-st.pem
2023/06/04 11:59:24 [INFO] Stored Issuer public key at /home/Numbers00/Personal/GitHub/sp-porkwatch/blockchain/infrastructure/sample-network/temp/enrollments/org1/users/arcilla@example.com/msp/IssuerPublicKey
2023/06/04 11:59:24 [INFO] Stored Issuer revocation public key at /home/Numbers00/Personal/GitHub/sp-porkwatch/blockchain/infrastructure/sample-network/temp/enrollments/org1/users/arcilla@example.com/msp/IssuerRevocationPublicKey
mv: cannot stat '/keystore/*_sk': No such file or directory
```

Figure 27: Registering a New User pt.2

While the user's data and password are already present in the ledger, for them to authenticate themselves, a user with administrative privileges from the same organization is to enroll said user using their organization number, e.g., 1 in the case of the breeder organization, their email, and their password. This generates the key and certificate files to be securely sent, alongside the file of the organization's TLS certificate, to the user for them to use for authentication.

B..13 Updating an Existing User

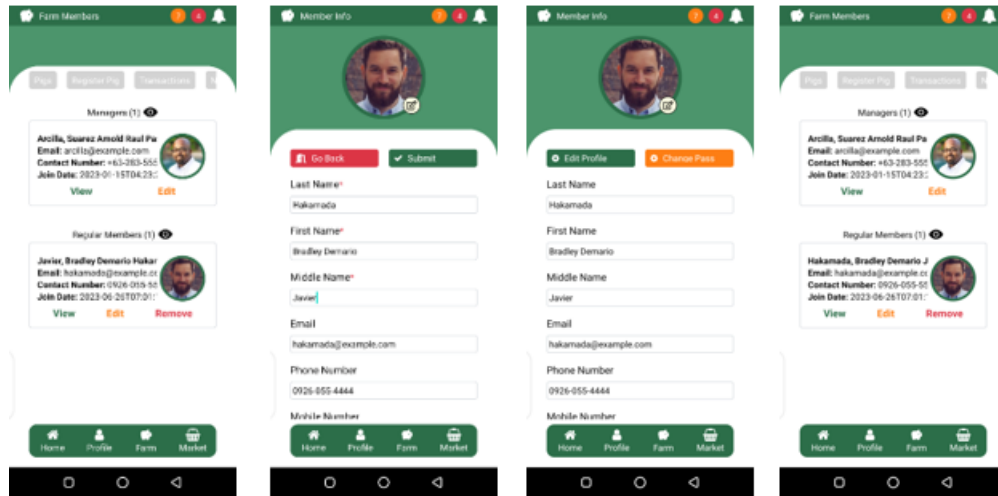


Figure 28: Updating an Existing User

Only managers from the same location can update the profile details of an existing user. For this, the manager would first navigate to the "Farm Members" screen. There, the manager clicks on the "Edit" link under the card of the user they wish to update the profile of. Once in the "Member Info" screen, the manager then clicks on the "Edit Profile" at the top left, before updating the fields and pressing on the "Submit" button at the top or bottom right sections to proceed with updating the user's profile. The manager is then redirected back to the view mode of the user's profile with the updated fields, indicating that the user's profile has been successfully updated.

B..14 Removing a User from a Location

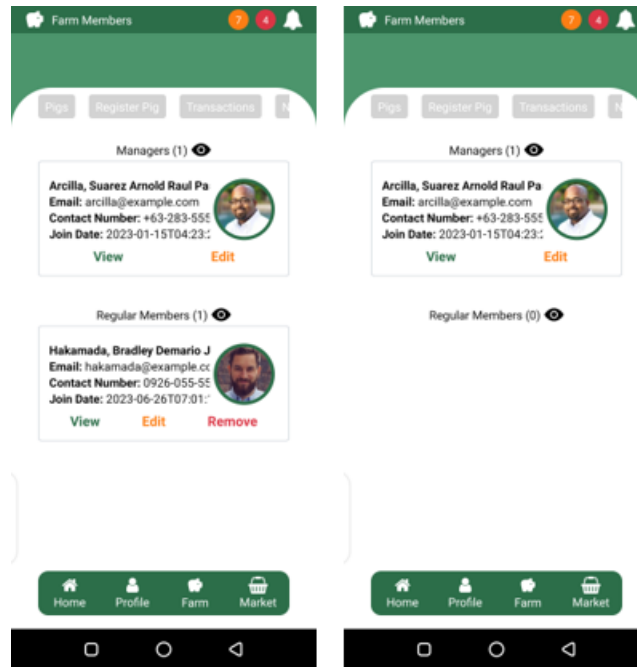


Figure 29: Removing a User from a Location

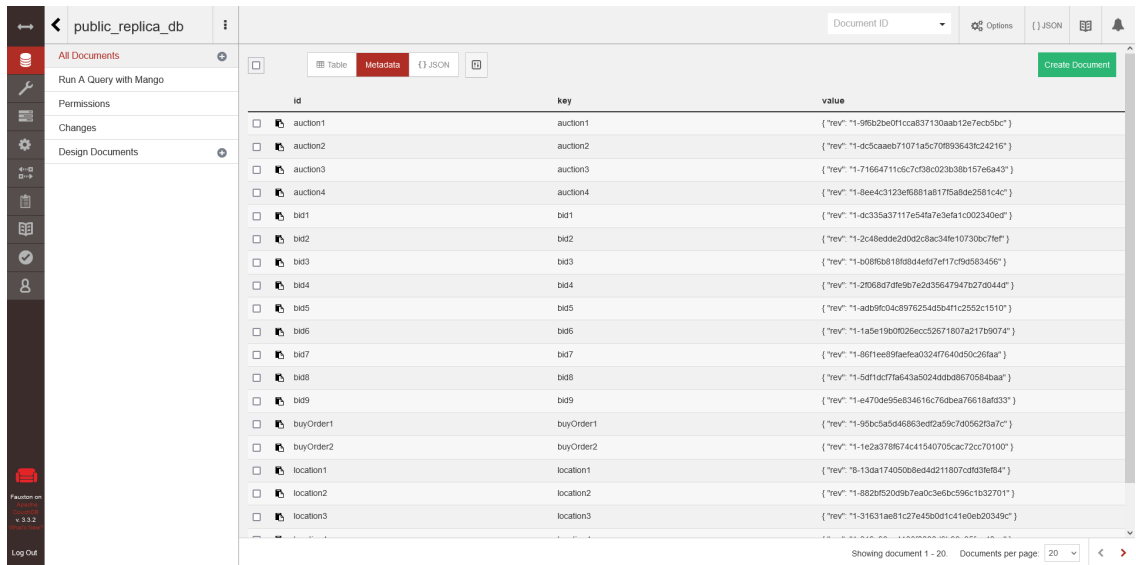
Only managers from the same location can remove a user from said location. For this, the manager would also first navigate to the "Farm Members" screen. There, the manager clicks on the "Remove" link under the card of the user they wish to remove from the location. The user, now without a location, would disappear from the "Farm Members" screen of all members in the location, and the user would be unable to authenticate, necessitating undergoing a new registration process if they are, for instance, to be reassigned to another location.

C. Public API Service



Name	Size	# of Docs	Partitioned	Actions
<code>_replicator</code>	0 bytes	0	No	  
<code>_users</code>	2.3 KB	1	No	  
<code>public_replica_db</code>	33.5 KB	59	No	  

Figure 30: Dockerized Off-chain CouchDB



id	key	value
auction1	auction1	{ "rev": "1-99b2be01cca837130aab12e7ec5b0c" }
auction2	auction2	{ "rev": "1-dc5caae71071a5c70899643c24216" }
auction3	auction3	{ "rev": "1-71664711c6c7c38c023b38b157e6aa43" }
auction4	auction4	{ "rev": "1-8ee4c3123e6681a8175a8de2581c4c" }
bid1	bid1	{ "rev": "1-dc335a37117e54fa7e3efa1c002340ed" }
bid2	bid2	{ "rev": "1-2c48edde2d0d2c8ac34fe10730bc7fe7" }
bid3	bid3	{ "rev": "1-40862b18f8d4e97e17c79d583456" }
bid4	bid4	{ "rev": "1-2068d7dfe9b7e2d335647947b279044d" }
bid5	bid5	{ "rev": "1-adb9fc04e976254d5b4f1c2552c1510" }
bid6	bid6	{ "rev": "1-1a5e1960026ecc32671807a217b9074" }
bid7	bid7	{ "rev": "1-86f1ee89faea0324764d450c26faa" }
bid8	bid8	{ "rev": "1-5d11dc77a643a5024dddb8670564baa" }
bid9	bid9	{ "rev": "1-e470a95e834616c76d8ea76618af033" }
buyOrder1	buyOrder1	{ "rev": "1-95bc5a5d46863ed2a59c7d05623a7c" }
buyOrder2	buyOrder2	{ "rev": "1-1e2a378f674c41540705cac72cc70100" }
location1	location1	{ "rev": "1-13da174050b6d4d211807c0d33f8e84" }
location2	location2	{ "rev": "1-882b52099b7ea0c36bc996c1b327011" }
location3	location3	{ "rev": "1-31631a681c27e45b0d1c41e0eb20349c" }

Figure 31: The Slave Database

The public API service is comprised of the off-chain slave CouchDB database and the secondary server hosting the controllers allowing for any user to perform rich queries into the system. The public API hosts the same GET endpoints as that of the main server, save for those related to passwords.

As stated in the system architecture overview, the public API service is kept up-to-date with the on-chain state databases through the main server populating the off-chain slave database with payloads processed from all events emitted in all the blockchain network's smart contracts. As the slave database is hosted off-chain, users

of the public API service benefits from the speed and convenience of conventional systems, making it useful for processes requiring the consumption of large amounts of data at once, such as for data analysis and for dashboards and visualizations.

User traffic to the public API service also does not degrade the performance of the blockchain network, given that both the secondary server and the slave database are hosted off-chain.

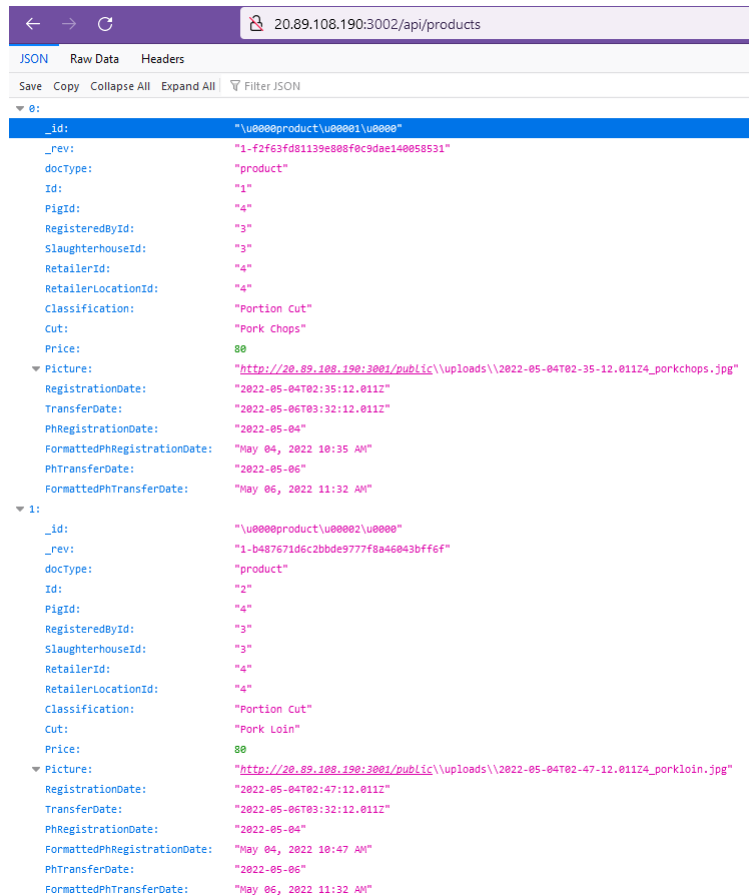


Figure 32: Requesting all Products in the System

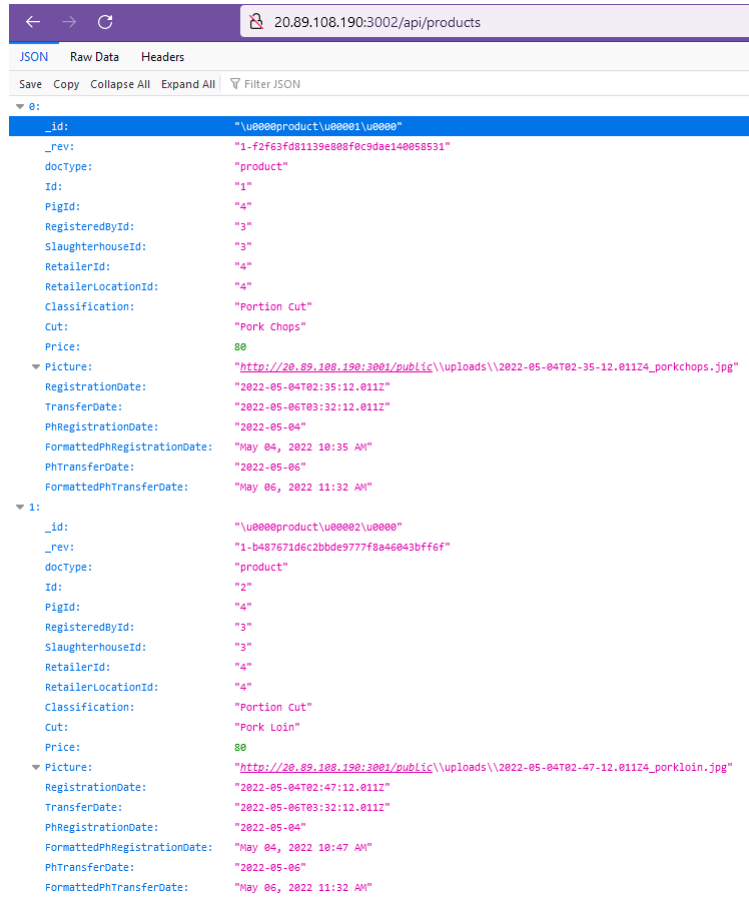


Figure 35: Using an URI-Encoded Querystring to Query Certain Products with Pagination

The above figures show some possible endpoint calls and queries for the products in the public API service. In Fig 32, the base endpoint of the products API is called, hence all products in the system are returned. In Fig 33, the product with id # 1 is asked, the controller on this endpoint returns a detailed version of the product in question, with the pig, slaughterhouse, etc. of the asset being returned as well. In Figs 34 and 35, the JSFiddle website is used to construct an URI-encoded stringified CouchDB query, this URI string is attached at the end of the /query endpoint of the public API to send a rich query to the server, here asking for the product with the cut of bacon in retailer location id # 4, with pagination.

VI. Discussion

The developed system, here dubbed the PorkWatch system for convenience, consisting of a blockchain system, a client mobile app, two servers, and an off-chain slave database, has fulfilled all of the main research objectives of the paper, that is, data immutability, data integrity, whole-of-chain traceability, automation, security, and reliability.

The system inherently benefits from data immutability through its use of blockchain technology in storing data. The blockchain system's distributed ledger consists of two main parts, the blockchain component and the state database component. The blockchain component is where transactions ordered in blocks are stored, and as blocks can only be appended to it, data, once added, cannot be changed. The on-chain state database serves to store the latest value of each data, or asset as they are called in Hyperledger Fabric. The PorkWatch system rarely needs to traverse the blockchain component of the ledger, getting the history of a pig or product is one of the few reasons for such traversals to happen, hence the system is able to stay performant, almost at the same level as conventional counterparts, while benefiting from blockchain technology.

Data integrity is fulfilled through the majority endorsement rule of the channel configuration. In the PorkWatch system, each transaction request by the client that can modify the data on the ledger, requires the endorsement, that is, the validation and approval of a majority of the peer organizations in the blockchain network, here requiring at least 3 out of 4 organizations to approve the request. The validation and subsequent ordering and committing processes are automatic, and does not require the intervention of any user. This independent validation of the transaction request from a majority of the peer nodes serve to make sure that data is maintained consistent and valid throughout the system, and that the codebase are up-to-date and kept consistent among at least a majority of the organizations, at least for the parts relevant to the

transaction.

Whole-of-chain traceability is achieved through the blockchain part of the distributed ledger. The system is able to reliably track all modifications and transfers affecting each pig or product asset through this immutable store of values. All users are able to download the entire history of a pig/product asset, from their registration in a breeder farm, to their final destinations, with every modification to the pig/product asset specified at any point in time.

Automation is achieved through a combination of client-based components, server-side controllers, and smart contracts hosted by the peer nodes in the blockchain network. These serve to represent certain real-world processes like confirming pig transfers, or approving bids, with processing and validation happening at essentially three layers for each transaction.

Security, that is, protecting the data from any sort of tampering, is achieved on both the private and permissioned blockchain system and the transparent and accessible public API service. On the blockchain-enabled side, that is, including the client mobile app that gets data directly from the tamperproof blockchain system, the majority endorsement rule and independent validation of peer nodes for each transaction request serves to protect the data from unilateral and malicious changes from even amongst its own member peer organizations, so long as, in this case, only a maximum of 2 out of 4 peer organizations are compromised.

On the side of the public API service, which has a second server querying data from the off-chain slave database, while the data there can be vulnerable to tampering with owing to its conventional implementation, these events can be easily remedied with a removal of the **checkpoint.json** file. This file serves to mark the latest block that was read by the event listener in the main server, and should this be removed, all emitted events from all blocks will be processed again. As this event listener serves to provide the off-chain slave database with non-confidential data from the distributed

ledger, the off-chain slave database and said event listener can simply be reset for the slave database to be repopulated from scratch, removing the effects of any tampering.

Lastly for the objectives, reliability is fulfilled by having each peer node host a copy of the chaincode and ledger. In a real production environment, with there being 4 peer organizations, each managing 2 peer nodes, the peer nodes will be separately hosted on different machines. So long as a peer organization has a peer node still operational, its members can still interact with the blockchain, and its remaining peer node can still participate in the endorsement process. Should all of the relevant machines of a single peer organization go down, the operation of the PorkWatch system will not halt. This means that if all 8 peer nodes are hosted on eight separate machines, a maximum of 5 peer nodes can go down, and so long as at least 3 organizations has a peer node left, little effect on the PorkWatch system can be perceived.

As the PorkWatch system comprises of essentially five distinct main components, both research and development took extensive time to accomplish. While Hyperledger Fabric is dominant in the realm of private permissioned blockchain platforms, especially for large enterprises, e.g. Walmart, Hitachi, IBM, and so on, its steep learning curve, extensive use of containerization technology, high complexity in setup, alongside its heavy toll on cpu, memory, and storage resources, makes it the most difficult aspect of the project to work with.

The client mobile app, having been developed using React Native Expo, also proved difficult to work with, with its relatively weaker libraries, being prone to sudden breakdowns, unhelpful error messages, and frequent dependency problems.

The main REST API server, acting as separate validator and intermediary between the client mobile app, the blockchain network, and even the off-chain slave database, also took a lot more time to develop than expected. The system required that the main server be able to handle inputs from the mobile application, including URI encoded query strings, validate the inputs, transform the inputs into a format

readable by the chaincode, call the appropriate function in the appropriate smart contract, handle a lot of possible errors, listen for events emitted from the blockchain, and so on. It required a lot of trial and error to get the main server to reliably act as intermediary between the client mobile app and the blockchain system.

The PorkWatch system meanwhile drew significant inspiration and reference from the paper titled "A blockchain-based multisignature approach for supply chain governance: A use case from the Australian beef study." [1] Specifically, the paper provided the inspiration for using blockchain technology to transform the supply chain governance in the Philippine pig industry, and for using a multisignature and proof-of-authority (PoA)-based approach to validating transaction requests and confirming the transfer of asset between two parties at each stage of the supply chain.

The PorkWatch system builds on their idea and approach on three fronts: (i) by using the enterprise-grade and open-source Hyperledger Fabric blockchain framework for the blockchain aspect of the system, (ii) by extending the use of blockchain technology in the system from only tracking the flow of livestock, to tracking multiple kinds of assets, generalizing the flow of livestock, i.e., pigs, and including representation of the location handling the pigs themselves as assets in the ledger, such that the system is able to accommodate the entry and removal of locations, and differences in the routes taken by the pigs from breeders to retailers, and (iii) by extending the private and permissioned blockchain network in the system with a transparent and accessible public API service, granting the public and auditors easy access to non-confidential data without the need for interacting with the blockchain itself, further enhancing the performance of the system and increasing its capacity.

The [full-stack-asset-transfer-guide](#) sample repository was also greatly helpful in establishing the infrastructure of the blockchain component of the PorkWatch system. The tutorial repository comes ready with just recipes and extensive shell codes and configuration files such that aside from developing the smart contracts, streamlining

the deployment process with custom shell files, the CouchDB indexes, and customizing the blockchain infrastructure through the configuration files, not much else was needed to set up and modify the blockchain network whenever and wherever deemed necessary.

VII. Conclusion

The PorkWatch system, consisting of a blockchain system, a client mobile app, a main backend server, a public API service, and an off-chain slave database, serves to transform the supply chain governance in the Philippine pig industry through blockchain technology, while maintaining, to an extent, the performance and convenience of conventional systems, and without the transaction costs common in blockchain-enabled systems.

The blockchain system was developed using Hyperledger Fabric, an enterprise-grade and open-source blockchain framework that is known for its high modularity and scalability. The blockchain component of the system serves to fulfill much of the main research objectives in this paper through the following:

- **Data Immutability** - All asset data are stored in the on-chain ledger, with its blockchain component serving to store the history of transactions and asset data in an immutable manner.
- **Data Integrity** - With a majority endorsement rule enforced in the blockchain network's channel configuration, a majority of the peer nodes will have to independently simulate and verify each transaction request and output, ensuring that data remains consistent throughout the system.
- **Whole-of-chain Traceability** - The entire history of any asset can be queried anytime, in the process traversing the blockchain component of the on-chain ledger. This data includes the initial state of the asset, and each subsequent modification until the present.
- **Automation** - Automation is achieved through the mobile app components, the controllers in the main server, and the smart contracts in the blockchain network, abstracting away much of the complexity of the system from the user's

perspective.

- **Security** - Data is ultimately secure on the blockchain network because of the blockchain component of the on-chain ledger. While the data on the off-chain slave database is not tamperproof, any effects can be easily remedied by a reset on the part of the slave database and the event listener on the main server, effectively overwrite the data on the database with up-to-date values from the secure blockchain network.
- **Reliability** - With the system having 4 peer organizations, each with 2 peer nodes, the system is able to handle the loss of an organization and up to an additional 3 other peer nodes before operation on the system is halted. The distributed nature of the ledger and chaincode means that they benefit from this redundancy of peer nodes, preventing a single point of failure.

As the blockchain system and its related permissions are constrained by the blockchain network's private and permissioned nature, transparency to the system is enabled by the off-chain slave database and the public API service, allowing the public and external auditors access to the system's data with the speed and convenience of a conventional system without needing the credentials to actually interact with the blockchain network.

VIII. Recommendations

While the PorkWatch system fulfilled its main research objectives, there is much to improve on some of many of its components, particularly in reducing overheads and for indexes to accommodate the mobile application's more complex queries. The system also lacks a dashboard and a benchmarking component, which would both prove very useful to the study's aim to explore blockchain technology's potential for provision of supply chain governance to the country's pig supply chains.

The blockchain system lacks a GUI for monitoring its transactions and components. The two known compatible systems to enable monitoring of transactions in the blockchain network are Hyperledger FireFly and Hyperledger Explorer. While the former is more recent, up-to-date, and powerful, setting it up would require many changes in the already streamlined deployment process available in the blockchain component of the system. Hyperledger Explorer is theoretically easier to set up, but due to its heavily outdated dependencies owing to an end-of-life status as of 2022, there was little luck in incorporating Hyperledger Explorer into the system.

The Hyperledger Fabric Operations Console, a web-based interface for managing the blockchain network's components, is deployed as a Kubernetes pod as part of the streamlined blockchain network setup process, but there were problems encountered with the scripts used to export the blockchain components, e.g., the peers and orderers, to JSON files for importing to the Fabric Console. Setting up the Fabric Console would allow for a more holistic view of the blockchain network, and would allow for further customization to its components through a GUI instead of being restricted to the CLI.

The client mobile app, while aiming to transform supply chain governance in the Philippine pig industry, was developed with little direct consultation from related experts and the actual prospective clients. The system relegates the health and quality checks on the pigs and product as a field that can be updated at any point of

the supply chain, this might not reflect actual systems in place for quality control in the Philippines.

The mobile app also grew far larger in complexity and scope than expected. Currently, the mobile app is completely reliant on the blockchain system for data, but much could be relegated to the conventional off-chain slave database for queries. In this case, only write operations would remain in all smart contracts in the blockchain network, save for the passwords smart contract, as no password is and should be moved to the off-chain slave database. Once this is implemented, screens in the mobile app would load much faster, and write operations will also significantly speed up, as certain write operations in the main server, such as for determining the id of a newly created asset, would make use of the slave database instead of the on-chain state databases, greatly reducing overhead in write operations.

Currently, no CouchDB index is deployed on the slave database. Save for the password index, all other indexes should be moved to the off-chain slave database once queries from the main server are moved to said database. The 29 indexes currently deployed on-chain are observed to be of no help to many of the queries from the mobile app, creating new indexes specifically for those queries would greatly speed up both read and write operations as well.

This paper also recommends developing a web-based application to serve as a publicly accessible dashboard for the system. This website can access the system's data through the public API service, which has already been developed. Hyperledger Caliper is also recommended to benchmark the system's performance, particularly its Transactions Per Second (TPS) and latency, at various levels of traffic, and for different use cases. It is recommended that such benchmarking be done after all other optimizations mentioned above, especially with regards to relegating queries to the off-chain slave database, are implemented, so as to measure the system's performance once many of the current unnecessary overheads are removed.

For use in an actual production setting, valid tokens should also be stored somewhere, possibly off-chain as access tokens are valid for merely an hour, to keep track of them. All smart contracts on the blockchain also currently receives the transaction's details through the user's id sent by the main server, which it decodes from the tokens. Although this process protects the system from identity-related issues arising from the mobile application, the blockchain itself is vulnerable to internal attacks, as administrators with access to the CLI can simply pass in any user's id and mask their identity, should they have access to said user's MSP credentials, i.e., their key, certificate, and their organization's TLS certificate, without needing to know said user's email and password.

Although technically, the MSP credentials are enough to protect identities on the blockchain, passwords serve as a second layer of protection should the MSP credentials be compromised. Circumventing them lessens security on the blockchain system. For this, the blockchain should instead also receive the undecoded JWT token and decode it in the smart contract itself. This way, the blockchain is better protected from internal attacks, as the token has to be signed on the main server, and the user should undergo the normal login process for a valid token to be created and signed.

IX. Bibliography

- [1] S. Cao, M. Foth, W. Powell, T. Miller, and M. Li, “A blockchain-based multisig-nature approach for supply chain governance: A use case from the australian beef industry,” *Blockchain: Research and Applications*, vol. 3, no. 4, p. 100091, 2022.
- [2] N. E. P. Manipol, M. S. Flores, R. Tan, N. Aquino, and G. Baticados, “Value chain analysis of philippine native swine (*sus scrofa philippinensis*) processed as lechon in major production areas in the philippines,” *Journal of Global Business and Trade*, no. 1, pp. 77–91, 2014.
- [3] M. Rola-Rubzen, F. Gabunada, and R. Mesorado, “Marketing systems for small livestock in the philippines: The case of western leYTE.” Available at https://www.researchgate.net/publication/254385418_Marketing_Systems_for_Small_Livestock_in_the_Philippines_The_Case_of_Western_Leyte (2002).
- [4] Statista, “Forecasted domestic volume of pork consumption in the philippines from january 2019 to january 2022.” Available at <https://www.statista.com/statistics/1170386/philippines-domestic-consumption-of-swine-meat/> (Accessed October, 2022), 2022.
- [5] D. L. Z. Cabantac, “Swine industry prospect in the philippines.” Available at https://rr-asia.woah.org/wp-content/uploads/2020/03/3-3-swine-industry-prospect_cabantac.pdf (Accessed October, 2022), 2018.

- [6] NationMaster, “Top countries in pork production - source oecd.” Available at <https://www.nationmaster.com/nmx/ranking/pork-production> (Accessed October, 2022), 2019.
- [7] J. P. Ayomen and M. S. Kingan, “Value chain analysis of pig (*sus scrota*) in a highland, indigenous community: The case of sablan, benguet, philippines,” *Mountain Journal of Science and Interdisciplinary Research*, vol. 79, no. 2, pp. 139–151, 2019.
- [8] OEC, “Pig meat in philippines.” Available at <https://oec.world/en/profile/bilateral-product/pig-meat/reporter/phl> (Accessed October, 2022), 2020.
- [9] D. C. Group, “Guidelines on mav plus pork import approved.” Available at <https://www.da.gov.ph/guidelines-on-mav-plus-pork-import-approved/> (Accessed October, 2022), 2021.
- [10] G. Yan, “Pork remains the favourite in the philippines.” Available at <https://www.thepigsite.com/articles/pork-remains-the-favourite-in-the-philippines> (Accessed December, 2022), 2020.
- [11] V. ter Beek, “Changing the mindset of philippine pig farms.” Available at <https://www.pigprogress.net/home/changing-the-mindset-of-philippine-pig-farmers/> (Accessed December, 2022), 2015.
- [12] R. M. Briones and I. B. Espineli, “Towards competitive livestock, poultry, and dairy industries: Consolidated benchmarking study.” Available at <https://www.pids.gov.ph/publication/discussion-papers/>

[towards-competitive-livestock-poultry-and-dairy-industries-consolidated-benchmark](#)
(Accessed December, 2022), 2022.

- [13] B. World, “Farm logistics.” Available at <https://www.pressreader.com/philippines/business-world/20200720/281852940880599> (Accessed October, 2022), 2020.
- [14] A. Parmentola, A. Petrillo, I. Tutore, and F. De Felice, “Is blockchain able to enhance environmental sustainability? a systematic review and research agenda from the perspective of sustainable development goals (sdgs),” *Business Strategy and the Environment*, vol. 31, no. 1, pp. 194–217, 2021.
- [15] D. C. Despres, “Models for supply chain governance,” *Proceedings of the 7th European Conference on Management Leadership and Governance*, pp. 535–537, 2011.
- [16] O. R. Young, “The effectiveness of international institutions: hard cases and critical variables,” in *Governance without Government Order and Change in World Politics* (E.-O. C. James N. Rosenau, ed.), ch. 6, Cambridge, UK: Cambridge University Press, 1992.
- [17] R. Eltantawy, “Supply chain governance role in supply chain risk management and sustainability,” in *Supply Chain Management* (S. Renko, ed.), ch. 19, Rijeka: Interchopen, 2011.
- [18] Z. Cao and F. Lumineau, “Revisiting the interplay between contractual and relational governance: A qualitative and meta-analytic investigation,” *Journal of Operations Management*, vol. 33-34, pp. 15–42, 2015.
- [19] H. Natarajan, S. Krause, and H. Gradstein, “Distributed ledger technology and blockchain,” World Bank Publications - Reports 29053, The World Bank Group, 2017.

- [20] M. Iansiti and K. R. Lakhani, “The truth about blockchain.” Available at <https://hbr.org/2017/01/the-truth-about-blockchain> (2017).
- [21] B. Ilsoe, “The internet of value: The controversies and opportunities in crypto technologies.” Available at <https://www.forbes.com/sites/forbesbusinesscouncil/2022/02/02/the-internet-of-value-the-controversies-and-opportunities-in-crypto-technology/?sh=302e7240110e> (2022).
- [22] M. Crosby, Nachiappan, P. Pattanayak, S. Verma, and V. Kalyanaraman, “Blockchain technology beyond bitcoin.” Available at <https://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf> (2015).
- [23] D. Ravi, S. Ramachandran, R. Vignesh, V. R. Falmari, and M. Brindha, “Privacy preserving transparent supply chain management through hyperledger fabric,” *Blockchain: Research and Applications*, vol. 3, no. 2, p. 100072, 2022.
- [24] S. L. Bager, C. Singh, and U. M. Persson, “Blockchain is not a silver bullet for agro-food supply chain sustainability: Insights from a coffee case study,” *Current Research in Environmental Sustainability*, vol. 4, p. 100163, 2022.
- [25] A. Sarfaraz, R. K. Chakraborty, and D. L. Essam, “The implications of blockchain-coordinated information sharing within a supply chain: A simulation study,” *Blockchain: Research and Applications*, p. 100110, 2022.
- [26] K. Shuaib, J. Abdella, F. Sallabi, and M. A. Serhani, “Secure decentralized electronic health records sharing system based on blockchains,” *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 8, Part A, pp. 5045–5058, 2022.

- [27] Z. Liao, J. Ai, S. Liu, Y. Zhang, and S. Liu, “Blockchain-based mobile crowdsourcing model with task security and task assignment,” *Expert Systems with Applications*, vol. 211, p. 118526, 2022.

X. Appendix

A. Smart Contracts

```
import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from 'fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination,
  getBid, getLocation, getPig, getUser
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Auction, DetailedAuction } from './models/auction';
import { Bid } from './models/bid';
import { Location } from './models/location';
import { Pig } from './models/pig';
import { User } from './models/user';

import sampleAuctions from './samples/auction';

export class AuctionContract extends Contract {

  constructor () {
    super('org.porkwatch.auction');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting(transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /^d+$/ .test(param));
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g. req.user.Id, is necessary for all transactions updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true) as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate details do not match!');
  }

  async unknownTransaction (ctx:Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().fcn;
    throw new Error('Unknown transaction function: ${transactionName}');
  }

  @Transaction()
  @Param('state', 'Auction', 'Part formed JSON of Auction')
  async CreateAuction (ctx: Context, state: Auction, transactorId: string): Promise<string> {
    const transactor = await retrieveUser(ctx, transactorId, true) as User;
    if (transactor.Id !== state.SellerId)
      throw new Error('The User with Id: ${transactor.Id} is not the seller at the Auction');

    const exists = await this.AuctionExists(ctx, state.Id);
```

```
    if (exists)
      throw new Error('The Auction with Id: ${state.Id} already exists');

    const createdAuction = Auction.newInstance(state);
    const createdAuctionBytes = marshal(createdAuction);

    const auctionKey = this.CreateAuctionKey(ctx, createdAuction.Id);
    await ctx.stub.putState(auctionKey, createdAuctionBytes);

    const indexes = [
      {
        name: 'auction_pigId',
        fields: [createdAuction.PigId]
      },
      {
        name: 'auction_sellerLocationId',
        fields: [createdAuction.SellerLocationId]
      },
      {
        name: 'auction_startDate',
        fields: [createdAuction.StartDate]
      }
    ];

    for (let i = 0; i < indexes.length; i++) {
      const fields = indexes[i].fields.map((field) => field.toString() || 'null');
      const indexKey = ctx.stub.createCompositeKey(indexes[i].name, fields);
      // Save index entry to state. Only the key name is needed, no need to store a duplicate copy of the marble.
      // Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null character as value
      await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
    }

    const eventPayload = marshal({
      ...createdAuction,
      key: auctionKey
    });
    ctx.stub.setEvent('CreateAuction', eventPayload);

    return stringify(createdAuction);
  }

  @Transaction()
  @Param('state', 'string', 'Part formed JSON of Auction')
  async CreateAuctionWithStringState (ctx: Context, state: string, transactorId: string): Promise<string> {
    let returnedAuction = '';
    try {
      const auction = JSON.parse(state) as Auction;
      returnedAuction = await this.CreateAuction(ctx, auction, transactorId);
    } catch (err) {
      console.log(err);
    }
    return returnedAuction;
  }

  @Transaction()
  @Param('state', 'Auction', 'Part formed JSON of Auction')
  async UpdateAuction (ctx: Context, state: Auction, transactorId: string): Promise<string> {
    await retrieveUser(ctx, transactorId, true) as User;

    if (!state.Id)
      throw new Error('No auction Id provided');

    const existingAuction = unmarshal(await this.ReadAuction(ctx, state.Id)) as Auction;

    // 4th optional param here is an object that would override the changes from state being copied into existingAuction
    // for updatedAuction
    const updatedState = Object.assign({}, existingAuction, state);
    const updatedAuction = Auction.newInstance(updatedState);
    const updatedAuctionBytes = marshal(updatedAuction);

    // no need to do anything with the CouchDB indexes for update transactions

    const auctionKey = this.CreateAuctionKey(ctx, updatedAuction.Id);
    await ctx.stub.putState(auctionKey, updatedAuctionBytes);

    const eventPayload = marshal({
```

```

    ...updatedAuction,
    key: auctionKey
  });
  ctx.stub.setEvent('UpdateAuction', eventPayload);

  return stringify(updatedAuction);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of
  Auction')
async UpdateAuctionWithStringState (ctx: Context, state: string,
  transactorId: string): Promise<string> {
  let returnedAuction = '';
  try {
    const auction = JSON.parse(state) as Auction;
    returnedAuction = await this.UpdateAuction(ctx, auction,
      transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedAuction;
}

@Transaction(false)
@Returns('string')
async GetAuctionHistory (ctx: Context, id: string): Promise<
  string> {
  const results = await getAssetHistory(ctx, this.
    CreateAuctionKey(ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'auction'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultForQueryString(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllAuctions (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'auction'
    }
  });
  return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetAuction (ctx: Context, id: string, isDetailed?:boolean)
  : Promise<string> {
  const auction = unmarshal(await this.ReadAuction(ctx, id) as
    Auction);
  if (!auction || !Object.keys(auction))
    throw new Error('The Auction with Id: ${id} does not exist');
  if (!isDetailed)
    return stringify(auction);

  const pig = unmarshal(await getPig(ctx, auction.PigId)) as Pig
  ;
  const seller = unmarshal(await getUser(ctx, auction.SellerId))
    as User;
  const sellerLocation = unmarshal(await getLocation(ctx,
    auction.SellerLocationId)) as Location;
  const winningBid = auction.WinningBidId

  ? unmarshal(await getBid(ctx, auction.WinningBidId)) as Bid
  : null;

  let activeBids: Bid[] = [];
  for (const id of auction.ActiveBidIds) {
    const activeBid = unmarshal(await getBid(ctx, id)) as Bid;
    activeBids.push(activeBid);
  }

  let canceledBids: Bid[] = [];
  for (const id of auction.CanceledBidIds) {
    const activeBid = unmarshal(await getBid(ctx, id)) as Bid;
    canceledBids.push(activeBid);
  }

  let rejectedBids: Bid[] = [];
  for (const id of auction.RejectedBidIds) {
    const rejectedBid = unmarshal(await getBid(ctx, id)) as Bid;
    rejectedBids.push(rejectedBid);
  }

  const bidAcceptedBy = auction.BidAcceptedById
  ? unmarshal(await getUser(ctx, auction.BidAcceptedById)) as
    User
  : null;
  const canceledBy = auction.CanceledById
  ? unmarshal(await getUser(ctx, auction.CanceledById)) as User
  : null;

  const detailedAuction: DetailedAuction = {
    ...auction,
    Pig: pig,
    Seller: seller,
    SellerLocation: sellerLocation,
    WinningBid: winningBid,
    ActiveBids: activeBids,
    CanceledBids: canceledBids,
    RejectedBids: rejectedBids,
    BidAcceptedBy: bidAcceptedBy,
    CanceledBy: canceledBy
  }

  return stringify(detailedAuction);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data

  for await (const auction of sampleAuctions) {
    try {
      await this.CreateAuction(ctx, auction, auction.SellerId);
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('AuctionInitLedger', Buffer.from('Auction
    Ledger Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'auction'
    }
  });
  return correctedQueryString;
}

CreateAuctionKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('auction', [id]);
}

async ReadAuction (ctx: Context, id: string): Promise<Uint8Array
  > {
  return readAsset(ctx, this.CreateAuctionKey(ctx, id));
}

async AuctionExists (ctx: Context, id: string): Promise<boolean>
  {
  return assetExists(ctx, this.CreateAuctionKey(ctx, id));
}

import stringify from 'json-stringify-deterministic';

```

```

import { Context, Contract, Param, Returns, Transaction } from '
fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Auction } from './models/auction';
import { Bid, DetailedBid } from './models/bid';
import { Location } from './models/location';
import { Pig } from './models/pig';
import { User } from './models/user';

import sampleBids from './samples/bid';

export class BidContract extends Contract {

  constructor () {
    super('org.porkwatch.bid');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /^~\d+$/ .test(param))
      ;
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate
        details do not match!');
  }

  async unknownTransaction (ctx: Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().
      fcn;
    throw new Error('Unknown transaction function: ${
      transactionName}');
  }

  @Transaction()
  @Param('state', 'Bid', 'Part formed JSON of Bid')
  async CreateBid (ctx: Context, state: Bid, transactorId: string)
    : Promise<string> {
    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Id !== state.BuyerId)
      throw new Error('The User with Id: ${transactor.Id} is not
        the buyer of the Bid');

    const exists = await this.BidExists(ctx, state.Id);
    if (exists)
      throw new Error('The Bid with Id: ${state.Id} already exists
        ');

    const createdBid = Bid.newInstance(state);
    const createdBidBytes = marshal(createdBid);

    const bidKey = this.CreateBidKey(ctx, createdBid.Id);
    await ctx.stub.putState(bidKey, createdBidBytes);

    const indexes = [

```

```

      name: 'bid_pigId',
      fields: [createdBid.PigId]
    ],
    {
      name: 'bid_sellerLocationId',
      fields: [createdBid.SellerLocationId]
    },
    {
      name: 'bid_buyerLocationId',
      fields: [createdBid.BuyerLocationId]
    },
    {
      name: 'bid_offerDate',
      fields: [createdBid.OfferDate]
    }
  ]
};

for (let i = 0; i < indexes.length; i++) {
  const fields = indexes[i].fields.map((field) => field.
    toString() || 'null');
  const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
    fields);
  // Save index entry to state. Only the key name is needed, no
  // need to store a duplicate copy of the marble.
  // Note - passing a 'nil' value will effectively delete the
  // key from state, therefore we pass null character as
  // value
  await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
}

const eventPayload = marshal({
  ...createdBid,
  key: bidKey
});
ctx.stub.setEvent('CreateBid', eventPayload);

return stringify(createdBid);
}

@Transaction()
@Param('state', 'string', 'Part formed JSON of Bid')
async CreateBidWithStringState (ctx: Context, state: string,
  transactorId: string): Promise<string> {
  let returnedBid = '';
  try {
    const bid = JSON.parse(state) as Bid;
    returnedBid = await this.CreateBid(ctx, bid, transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedBid;
}

@Transaction()
@Param('state', 'Bid', 'Part formed JSON of Bid')
async UpdateBid (ctx: Context, state: Bid, transactorId: string)
  : Promise<string> {
  await retrieveUser(ctx, transactorId, true) as User;

  if (!state.Id)
    throw new Error('No bid Id provided');

  const existingBid = unmarshal(await this.ReadBid(ctx, state.Id)
    ) as Bid;

  // 4th optional param here is an object that would
  // override the changes from state being copied into
  // existingBid
  // for updatedBid
  const updatedState = Object.assign({}, existingBid, state);
  const updatedBid = Bid.newInstance(updatedState);
  const updatedBidBytes = marshal(updatedBid);

  // no need to do anything with the CouchDB indexes for update
  // transactions

  const bidKey = this.CreateBidKey(ctx, updatedBid.Id);
  await ctx.stub.putState(bidKey, updatedBidBytes);

  const eventPayload = marshal({
    ...updatedBid,
    key: bidKey
  });
  ctx.stub.setEvent('UpdateBid', eventPayload);

  return stringify(updatedBid);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of Bid')
async UpdateBidWithStringState (ctx: Context, state: string,
  transactorId: string): Promise<string> {

```

```

let returnedBid = '';
try {
  const bid = JSON.parse(state) as Bid;
  returnedBid = await this.UpdateBid(ctx, bid, transactionId);
} catch (err) {
  console.log(err);
}
return returnedBid;
}

@Transaction(false)
@Returns('string')
async GetBidHistory (ctx: Context, id: string): Promise<string>
{
  const results = await getAssetHistory(ctx, this.CreateBidKey(
    ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'bid'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultForQueryString(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllBids (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'bid'
    }
  });
  return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetBid (ctx: Context, id: string, isDetailed?:boolean):
Promise<string> {
  const bid = unmarshal(await this.ReadBid(ctx, id)) as Bid;
  if (!bid || !Object.keys(bid))
    throw new Error('The Bid with Id: ${id} does not exist');

  if (!isDetailed)
    return stringify(bid);

  const auction = unmarshal(await getAuction(ctx, bid.AuctionId)
    ) as Auction;
  const pig = unmarshal(await getPig(ctx, bid.PigId)) as Pig;
  const seller = unmarshal(await getUser(ctx, bid.SellerId)) as
    User;
  const sellerLocation = unmarshal(await getLocation(ctx, bid.
    SellerLocationId)) as Location;
  const buyer = unmarshal(await getUser(ctx, bid.BuyerId)) as
    User;
  const buyerLocation = unmarshal(await getLocation(ctx, bid.
    BuyerLocationId)) as Location;
  const acceptedBy = bid.AcceptedById
    ? unmarshal(await getUser(ctx, bid.AcceptedById)) as User
    : null;
  const rejectedBy = bid.RejectedById
    ? unmarshal(await getUser(ctx, bid.RejectedById)) as User
    : null;
  const canceledBy = bid.CanceledById
    ? unmarshal(await getUser(ctx, bid.CanceledById)) as User
    : null;

  const detailedBid: DetailedBid = {
    ...bid,
    Auction: auction,
    Pig: pig,
    Seller: seller,
    SellerLocation: sellerLocation,
    Buyer: buyer,
    BuyerLocation: buyerLocation,
    AcceptedBy: acceptedBy,
    RejectedBy: rejectedBy,
    CanceledBy: canceledBy
  };

  return stringify(detailedBid);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data

  for await (const bid of sampleBids) {
    try {
      await this.CreateBid(ctx, bid, bid.BuyerId);
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('BidInitLedger', Buffer.from('Bid Ledger
    Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'bid'
    }
  });
  return correctedQueryString;
}

CreateBidKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('bid', [id]);
}

async ReadBid (ctx: Context, id: string): Promise<Uint8Array> {
  return readAsset(ctx, this.CreateBidKey(ctx, id));
}

async BidExists (ctx: Context, id: string): Promise<boolean> {
  return assetExists(ctx, this.CreateBidKey(ctx, id));
}

import stringify from 'json-stringify-deterministic';
import { Context, Contract, Param, Returns, Transaction } from '
  fabric-contract-api';
import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination
}
  ,
  getLocation, getProduct, getUser
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';
import { BuyOrder, DetailedBuyOrder } from './models/buyOrder';
import { Location } from './models/location';
import { Product } from './models/product';
import { User } from './models/user';
import sampleBuyOrders from './samples/buyOrder';
export class BuyOrderContract extends Contract {
  constructor () {
    super('org.porkwatch.buyorder');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();

```



```

const transactionName = funcAndParams.fcn;

console.log('transactionName', transactionName);
console.log('isTransactionSubmitting', isTransactionSubmitting
(transactionName));
console.log('clientCommonName', getClientCommonName(ctx));
if (transactionName.endsWith('InitLedger')) {
  // Skip custom logic for InitLedger transaction
  return;
}

const params = funcAndParams.params;
if (!isTransactionSubmitting(transactionName))
  return;

let transactorId = params.find((param) => /^d+$/ .test(param))
;
if (!transactorId)
  throw new Error('No transactorId found, transactorId, e.g.
req.user.Id, is necessary for all transactions
updating the ledger');

const transactor = await retrieveUser(ctx, transactorId, true)
as User;
if (transactor.Email !== getClientCommonName(ctx))
  throw new Error('User credentials and X.509 certificate
details do not match!');
}

async unknownTransaction (ctx:Context): Promise<void> {
  const transactionName = ctx.stub.getFunctionAndParameters().
  fcn;
  throw new Error('Unknown transaction function: ${
transactionName}');
}

@Transaction()
@Param('state', 'BuyOrder', 'Part formed JSON of BuyOrder')
async CreateBuyOrder (ctx: Context, state: BuyOrder,
  transactorId: string): Promise<string> {
  const transactor = await retrieveUser(ctx, transactorId, true)
  as User;
  if (transactor.Id !== state.RetailerId)
    throw new Error('The User with Id: ${transactor.Id} is not
    the initiator of the BuyOrder');

  const exists = await this.BuyOrderExists(ctx, state.Id);
  if (exists)
    throw new Error('The BuyOrder with Id: ${state.Id} already
    exists');

  const createdBuyOrder = BuyOrder.newInstance(state);
  const createdBuyOrderBytes = marshal(createdBuyOrder);

  const buyOrderKey = this.CreateBuyOrderKey(ctx,
  createdBuyOrder.Id);
  await ctx.stub.putState(buyOrderKey, createdBuyOrderBytes);

  const indexes = [
    {
      name: 'buyOrder_slaughterhouseId',
      fields: [createdBuyOrder.SlaughterhouseId]
    },
    {
      name: 'buyOrder_retailerLocationId',
      fields: [createdBuyOrder.RetailerLocationId]
    },
    {
      name: 'buyOrder_orderDate',
      fields: [createdBuyOrder.OrderDate]
    }
  ];

  for (let i = 0; i < indexes.length; i++) {
    const fields = indexes[i].fields.map((field) => field.
    toString() || 'null');
    const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
    fields);
    // Save index entry to state. Only the key name is needed, no
    need to store a duplicate copy of the marble.
    // Note - passing a 'nil' value will effectively delete the
    key from state, therefore we pass null character as
    value
    await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
  }

  const eventPayload = marshal({
    ...createdBuyOrder,
    key: buyOrderKey
  });
  ctx.stub.setEvent('CreateBuyOrder', eventPayload);

  return stringify(createdBuyOrder);
}

@Transaction()
@Param('state', 'string', 'Part formed JSON of BuyOrder')
async CreateBuyOrderWithStringState (ctx: Context, state: string
  , transactorId: string): Promise<string> {
  let returnedBuyOrder = '';
  try {
    const buyOrder = JSON.parse(state) as BuyOrder;
    returnedBuyOrder = await this.CreateBuyOrder(ctx, buyOrder,
    transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedBuyOrder;
}

@Transaction()
@Param('state', 'BuyOrder', 'Part formed JSON of BuyOrder')
async UpdateBuyOrder (ctx: Context, state: BuyOrder,
  transactorId: string): Promise<void> {
  await retrieveUser(ctx, transactorId, true) as User;

  if (!state.Id)
    throw new Error('No buyOrder Id provided');

  const existingBuyOrder = unmarshal(await this.ReadBuyOrder(ctx
  , state.Id) as BuyOrder);

  // 4th optional param here is an object that would
  // override the changes from state being copied into
  existingBuyOrder
  // for updatedBuyOrder
  const updatedState = Object.assign({}, existingBuyOrder, state
  );
  const updatedBuyOrder = BuyOrder.newInstance(updatedState);
  const updatedBuyOrderBytes = marshal(updatedBuyOrder);

  // no need to do anything with the CouchDB indexes for update
  transactions

  const buyOrderKey = this.CreateBuyOrderKey(ctx,
  updatedBuyOrder.Id);
  await ctx.stub.putState(buyOrderKey, updatedBuyOrderBytes);

  const eventPayload = marshal({
    ...updatedBuyOrder,
    key: buyOrderKey
  });
  ctx.stub.setEvent('UpdateBuyOrder', eventPayload);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of
  BuyOrder')
async UpdateBuyOrderWithStringState (ctx: Context, state: string
  , transactorId: string): Promise<void> {
  const buyOrder = JSON.parse(state) as BuyOrder;
  try {
    await this.UpdateBuyOrder(ctx, buyOrder, transactorId);
  } catch (err) {
    console.log(err);
  }
}

@Transaction(false)
@Returns('string')
async GetBuyOrderHistory (ctx: Context, id: string): Promise<
  string> {
  const results = await getAssetHistory(ctx, this.
  CreateBuyOrderKey(ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'buyOrder'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
  queryString);
  const results = unmarshal(await getQueryResultForQueryString(
  ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

```

```

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllBuyOrders (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'buyOrder'
    }
  });
  return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetBuyOrder (ctx: Context, id: string, isDetailed?:boolean
): Promise<string> {
  const buyOrder = unmarshal(await this.ReadBuyOrder(ctx, id))
    as BuyOrder;
  if (!buyOrder || !Object.keys(buyOrder))
    throw new Error('The BuyOrder with Id: ${id} does not exist')
    ;

  if (!isDetailed)
    return stringify(buyOrder);

  const slaughterhouse = unmarshal(await getLocation(ctx,
    buyOrder.SlaughterhouseId)) as Location;
  const retailer = unmarshal(await getUser(ctx, buyOrder.
    RetailerId)) as User;
  const retailerLocation = unmarshal(await getLocation(ctx,
    buyOrder.RetailerLocationId)) as Location;
  let products: Product[] = [];
  for (const id of buyOrder.ProductIds) {
    const product = unmarshal(await getProduct(ctx, id)) as
    Product;
    products.push(product);
  }

  const detailedBuyOrder: DetailedBuyOrder = {
    ...buyOrder,
    Slaughterhouse: slaughterhouse,
    Retailer: retailer,
    RetailerLocation: retailerLocation,
    Products: products
  };

  return stringify(detailedBuyOrder);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data

  for await (const buyOrder of sampleBuyOrders) {
    try {
      await this.CreateBuyOrder(ctx, buyOrder, buyOrder.
        RetailerId);
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('BuyOrderInitLedger', Buffer.from('Buy Order
    Ledger Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'buyOrder'
    }
  });
  return correctedQueryString;
}

CreateBuyOrderKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('buyOrder', [id]);
}

async ReadBuyOrder (ctx: Context, id: string): Promise<
  Uint8Array> {
  return readAsset(ctx, this.CreateBuyOrderKey(ctx, id));
}

async BuyOrderExists (ctx: Context, id: string): Promise<boolean
  > {
  return assetExists(ctx, this.CreateBuyOrderKey(ctx, id));
}

import stringify from 'json-stringify-deterministic';
import { Context, Contract, Param, Returns, Transaction } from '
  fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Location, DetailedLocation } from './models/location';
import { Pig } from './models/pig';
import { Product } from './models/product';
import { User } from './models/user';

import sampleLocations from './samples/location';

export class LocationContract extends Contract {
  constructor () {
    super('org.porkwatch.location');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /\d+$/ .test(param))
      ;
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate
        details do not match!');
  }

  async unknownTransaction (ctx:Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().
      fcn;
    throw new Error('Unknown transaction function: ${
      transactionName}');
  }

  @Transaction()
  @Param('state', 'Location', 'Part formed JSON of Location')
  async CreateLocation (ctx: Context, state: Location,
    transactorId: string): Promise<string> {
    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Id !== state.ManagerIds[0])
      throw new Error('The User with Id: ${transactor.Id} is not
        the manager of the Location');
  }
}

```

```

const exists = await this.LocationExists(ctx, state.Id);
if (exists)
  throw new Error('The Location with Id: ${state.Id} already
    exists');

const createdLocation = Location.newInstance(state);
const createdLocationBytes = marshal(createdLocation);

const locationKey = this.CreateLocationKey(ctx,
  createdLocation.Id);
await ctx.stub.putState(locationKey, createdLocationBytes);

const indexes = [
  {
    name: 'location_type',
    fields: [createdLocation.Type]
  },
  {
    name: 'location_registrationDate',
    fields: [createdLocation.RegistrationDate]
  }
];

for (let i = 0; i < indexes.length; i++) {
  const fields = indexes[i].fields.map((field) => field.
    toString() || 'null');
  const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
    fields);
  // Save index entry to state. Only the key name is needed, no
  // need to store a duplicate copy of the marble.
  // Note - passing a 'nil' value will effectively delete the
  // key from state, therefore we pass null character as
  // value
  await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
}

const eventPayload = marshal({
  ...createdLocation,
  key: locationKey
});
ctx.stub.setEvent('CreateLocation', eventPayload);

return stringify(createdLocation);
}

@Transaction()
@Param('state', 'string', 'Part formed JSON of Location')
async CreateLocationWithStringState (ctx: Context, state: string
  , transactorId: string): Promise<string> {
  let returnedLocation = '';
  try {
    const location = JSON.parse(state) as Location;
    returnedLocation = await this.CreateLocation(ctx, location,
      transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedLocation;
}

@Transaction()
@Param('state', 'Location', 'Part formed JSON of Location')
async UpdateLocation (ctx: Context, state: Location,
  transactorId: string): Promise<string> {
  await retrieveUser(ctx, transactorId, true) as User;

  if (!state.Id)
    throw new Error('No location Id provided');

  const existingLocation = unmarshal(await this.ReadLocation(ctx
    , state.Id)) as Location;

  // 4th optional param here is an object that would
  // override the changes from state being copied into
  // existingLocation
  // for updatedLocation
  const updatedState = Object.assign({}, existingLocation, state
  );
  const updatedLocation = Location.newInstance(updatedState);
  const updatedLocationBytes = marshal(updatedLocation);

  // no need to do anything with the CouchDB indexes for update
  // transactions

  const locationKey = this.CreateLocationKey(ctx,
    updatedLocation.Id);
  await ctx.stub.putState(locationKey, updatedLocationBytes);

  const eventPayload = marshal({
    ...updatedLocation,
    key: locationKey
  });
  ctx.stub.setEvent('UpdateLocation', eventPayload);

  return stringify(updatedLocation);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of
  Location')
async UpdateLocationWithStringState (ctx: Context, state: string
  , transactorId: string): Promise<string> {
  let returnedLocation = '';
  try {
    const location = JSON.parse(state) as Location;
    returnedLocation = await this.UpdateLocation(ctx, location,
      transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedLocation;
}

@Transaction(false)
@Returns('string')
async GetLocationHistory (ctx: Context, id: string): Promise<
  string> {
  const results = await getAssetHistory(ctx, this.
    CreateLocationKey(ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'location'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultForQueryString(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllLocations (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'location'
    }
  });
  return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetLocation (ctx: Context, id: string, isDetailed?:boolean
  ): Promise<string> {
  const location = unmarshal(await this.ReadLocation(ctx, id))
    as Location;
  if (!location || !Object.keys(location))
    throw new Error('The Location with Id: ${id} does not exist')
    ;

  if (!isDetailed)
    return stringify(location);

  let members: User[] = [];
  for (const id of location.MemberIds) {
    const member = unmarshal(await getUser(ctx, id)) as User;
    members.push(member);
  }

  let managers: User[] = [];
  for (const id of location.ManagerIds) {

```

```

    const manager = unmarshal(await getUser(ctx, id)) as User;
    managers.push(manager);
  }

  let pigs: Pig[] = [];
  for (const id of location.PigIds) {
    const pig = unmarshal(await getPig(ctx, id)) as Pig;
    pigs.push(pig);
  }

  let products: Product[] = [];
  for (const id of location.ProductIds) {
    const product = unmarshal(await getProduct(ctx, id)) as
      Product;
    products.push(product);
  }

  const detailedLocation: DetailedLocation = {
    ...location,
    Members: members,
    Managers: managers,
    Pigs: pigs,
    Products: products
  };

  return stringify(detailedLocation);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data

  for await (const location of sampleLocations) {
    try {
      await this.CreateLocation(ctx, location, location.
        ManagerIds[0]);
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('LocationInitLedger', Buffer.from('Location
    Ledger Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'location'
    }
  });
}

CreateLocationKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('location', [id]);
}

async ReadLocation (ctx: Context, id: string): Promise<
  Uint8Array> {
  return readAsset(ctx, this.CreateLocationKey(ctx, id));
}

async LocationExists (ctx: Context, id: string): Promise<boolean
  > {
  return assetExists(ctx, this.CreateLocationKey(ctx, id));
}
}

import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from '
  fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination
}
from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Bid } from './models/bid';
import { Notification, DetailedNotification } from './models/
  notification';
import { Transfer } from './models/transfer';
import { User } from './models/user';

import sampleNotifications from './samples/notification';

export class NotificationContract extends Contract {

  constructor () {
    super('org.porkwatch.notification');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /^~d+$/ .test(param))
      ;
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate
        details do not match!');
  }

  async unknownTransaction (ctx: Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().
      fcn;
    throw new Error('Unknown transaction function: ${
      transactionName}');
  }

  @Transaction()
  @Param('state', 'Notification', 'Part formed JSON of
    Notification')
  async CreateNotification (ctx: Context, state: Notification,
    transactorId: string): Promise<string> {
    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Id !== state.InitiatorId)
      throw new Error('The User with Id: ${transactor.Id} is not
        the initiator of the Notification');

    const exists = await this.NotificationExists(ctx, state.Id);
    if (exists)
      throw new Error('The Notification with Id: ${state.Id}
        already exists');

    const createdNotification = Notification.newInstance(state);
    const createdNotificationBytes = marshal(createdNotification);

    const notificationKey = this.CreateNotificationKey(ctx,
      createdNotification.Id);
    await ctx.stub.putState(notificationKey,
      createdNotificationBytes);
  }

  const indexes = [
    {
      name: 'notification_issuanceDate',
      fields: [createdNotification.IssuanceDate]
    }
  ];
}

for (let i = 0; i < indexes.length; i++) {
  const fields = indexes[i].fields.map((field) => field.
    toString() || 'null');
  const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
    fields);
  // Save index entry to state. Only the key name is needed, no
  // need to store a duplicate copy of the marble.
  // Note - passing a 'nil' value will effectively delete the

```

```

        key from state, therefore we pass null character as
        value
        await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
    }

    const eventPayload = marshal({
        ...createdNotification,
        key: notificationKey
    });
    ctx.stub.setEvent('CreateNotification', eventPayload);

    return stringify(createdNotification);
}

@Transaction()
@Param('state', 'string', 'Part formed JSON of Notification')
async CreateNotificationWithStringState (ctx: Context, state:
string, transactorId: string): Promise<string> {
    let returnedNotification = '';
    try {
        const notification = JSON.parse(state) as Notification;
        returnedNotification = await this.CreateNotification(ctx,
notification, transactorId);
    } catch (err) {
        console.log(err);
    }
    return returnedNotification;
}

@Transaction()
@Param('state', 'Notification', 'Part formed JSON of
Notification')
async UpdateNotification (ctx: Context, state: Notification,
transactorId: string): Promise<string> {
    await retrieveUser(ctx, transactorId, true) as User;

    if (!state.Id)
        throw new Error('No notification Id provided');

    const existingNotification = unmarshal(await this.
ReadNotification(ctx, state.Id)) as Notification;

    // 4th optional param here is an object that would
    // override the changes from state being copied into
    existingNotification
    // for updatedNotification
    const updatedState = Object.assign({}, existingNotification,
state);
    const updatedNotification = Notification.newInstance(
updatedState);
    const updatedNotificationBytes = marshal(updatedNotification);

    // no need to do anything with the CouchDB indexes for update
    transactions

    const notificationKey = this.CreateNotificationKey(ctx,
updatedNotification.Id);
    await ctx.stub.putState(notificationKey,
updatedNotificationBytes);

    const eventPayload = marshal({
        ...updatedNotification,
        key: notificationKey
    });
    ctx.stub.setEvent('UpdateNotification', eventPayload);

    return stringify(updatedNotification);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of
Notification')
async UpdateNotificationWithStringState (ctx: Context, state:
string, transactorId: string): Promise<string> {
    let returnedNotification = '';
    try {
        const notification = JSON.parse(state) as Notification;
        await this.UpdateNotification(ctx, notification, transactorId
);
    } catch (err) {
        console.log(err);
    }
    return returnedNotification;
}

@Transaction(false)
@Returns('string')
async GetNotificationHistory (ctx: Context, id: string): Promise
<string> {
    const results = await getAssetHistory(ctx, this.
CreateNotificationKey(ctx, id));
    return stringify(results);
}

}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
    const totalCount = unmarshal(await getCount(ctx, 'notification
'));
    return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
string): Promise<string> {
    const correctedQueryString = this.CorrectQueryString(
queryString);
    const results = unmarshal(await getQueryResultForQueryString(
ctx, correctedQueryString)) as AssetJsonRes[];
    return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
string): Promise<string> {
    const correctedQueryString = this.CorrectQueryString(
queryString);
    const results = unmarshal(await getQueryResultWithPagination(
ctx, correctedQueryString)) as AssetJsonRes[];
    return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllNotifications (ctx: Context): Promise<string> {
    const queryString = stringify({
        selector: {
            docType: 'notification'
        }
    });
    return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetNotification (ctx: Context, id: string, isDetailed?:
boolean): Promise<string> {
    const notification = unmarshal(await this.ReadNotification(ctx
, id)) as Notification;
    if (!notification || !Object.keys(notification))
        throw new Error('The Notification with Id: ${id} does not
exist');

    if (!isDetailed)
        return stringify(notification);

    const initiator = unmarshal(await getUser(ctx, notification.
InitiatorId)) as User;

    let recipients: User[] = [];
    for (const id of notification.RecipientIds) {
        const recipient = unmarshal(await getUser(ctx, id)) as User;
        recipients.push(recipient);
    }

    const bid = unmarshal(await getBid(ctx, notification.BidId))
as Bid;
    const transfer = unmarshal(await getTransfer(ctx, notification
.TransferId)) as Transfer;

    const detailedNotification: DetailedNotification = {
        ...notification,
        Initiator: initiator,
        Recipients: recipients,
        Bid: bid,
        Transfer: transfer
    }

    return stringify(detailedNotification);
}

@Transaction()
async InitLedger (ctx: Context) {
    // initialize the ledger with activity data

    for await (const notification of sampleNotifications) {
        try {
            await this.CreateNotification(ctx, notification,
notification.InitiatorId);
        } catch (err) {
            console.log(err);
        }
    }
}

```

```

    }

    ctx.stub.setEvent('NotificationInitLedger', Buffer.from('
      Notification Ledger Initialized'));
  }

  CorrectQueryString (queryString: string | QueryString): string {
    const parsedQueryString = typeof queryString === 'string'
      ? unmarshal(queryString) as QueryString
      : queryString;
    const correctedQueryString = stringify({
      ...parsedQueryString,
      selector: {
        ...parsedQueryString.selector,
        docType: 'notification'
      }
    });
    return correctedQueryString;
  }

  CreateNotificationKey (ctx: Context, id: string): string {
    return ctx.stub.createCompositeKey('notification', [id]);
  }

  async ReadNotification (ctx: Context, id: string): Promise<
    Uint8Array> {
    return readAsset(ctx, this.CreateNotificationKey(ctx, id));
  }

  async NotificationExists (ctx: Context, id: string): Promise<
    boolean> {
    return assetExists(ctx, this.CreateNotificationKey(ctx, id));
  }
}

import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from '
  fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getCount, getResultForQueryString
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Password } from './models/password';
import { User } from './models/user';

import samplePasswords from './samples/password';

export class PasswordContract extends Contract {

  constructor () {
    super('org.porkwatch.password');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /^d+$/ .test(param))
      ;
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate
        details do not match!');
  }

  async unknownTransaction (ctx:Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().
      fcn;
    throw new Error('Unknown transaction function: ${
      transactionName}');
  }

  @Transaction()
  @Param('state', 'Password', 'Part formed JSON of Password')
  async CreatePassword (ctx: Context, state: Password,
    transactorId: string): Promise<string> {
    await retrieveUser(ctx, transactorId, true) as User;

    const exists = await this.PasswordExists(ctx, state.Id);
    if (exists)
      throw new Error('The Password with Id: ${state.Id} already
        exists');

    const createdPassword = Password.newInstance(state);
    const createdPasswordBytes = marshal(createdPassword);

    const passwordKey = this.CreatePasswordKey(ctx,
      createdPassword.Id);
    await ctx.stub.putState(passwordKey, createdPasswordBytes);

    const indexes = [
      {
        name: 'password_userId',
        fields: [createdPassword.UserId]
      }
    ];

    for (let i = 0; i < indexes.length; i++) {
      const fields = indexes[i].fields.map((field) => field.
        toString() || 'null');
      const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
        fields);
      // Save index entry to state. Only the key name is needed, no
        need to store a duplicate copy of the marble.
      // Note - passing a 'nil' value will effectively delete the
        key from state, therefore we pass null character as
        value
      await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
    }

    this.ExpirePassword(ctx, state.UserId, transactorId);

    return stringify(createdPassword);
  }

  @Transaction()
  @Param('state', 'string', 'Part formed JSON of Password')
  async CreatePasswordWithStringState (ctx: Context, state: string
    , transactorId: string): Promise<string> {
    let returnedPassword = '';
    try {
      const password = JSON.parse(state) as Password;
      returnedPassword = await this.CreatePassword(ctx, password,
        transactorId);
    } catch (err) {
      console.log(err);
    }
    return returnedPassword;
  }

  @Transaction()
  @Param('state', 'Password', 'Part formed JSON of Password')
  async ExpirePassword (ctx: Context, userId: string, transactorId
    : string): Promise<void> {
    await retrieveUser(ctx, transactorId, true) as User;

    if (!userId)
      throw new Error('No user Id provided');

    const queryString = stringify({
      selector: {
        docType: 'password',
        UserId: userId,
        IsActive: true
      },
      sort: [
        { Id: 'desc' }
      ],
      // Get the previous active password
      skip: 1,
      limit: 1
    });

    // If empty, it's the user's first password and therefore this
      should not run
    const prevPassword = unmarshal(await this.

```

```

    GetQueryResultForQueryString(ctx, queryString) as
    Password;
    if (!prevPassword || !Object.keys(prevPassword).length)
    return;

    const expiredPassword = {
      ...prevPassword,
      IsActive: false
    };
    const expiredPasswordBytes = marshal(expiredPassword);

    // no need to do anything with the CouchDB indexes for update
    transactions

    const passwordKey = this.CreatePasswordKey(ctx,
      expiredPassword.Id);
    await ctx.stub.putState(passwordKey, expiredPasswordBytes);
  }

  @Transaction(false)
  @Returns('string')
  async GetCount (ctx: Context): Promise<string> {
    const totalCount = unmarshal(await getCount(ctx, 'password'));
    return stringify(totalCount);
  }

  @Transaction(false)
  @Returns('string')
  async GetQueryResultForQueryString (ctx: Context, queryString:
    string): Promise<string> {
    const correctedQueryString = this.CorrectQueryString(
      queryString);
    const results = unmarshal(await getQueryResultForQueryString(
      ctx, correctedQueryString)) as AssetJsonRes[];
    return stringify(results);
  }

  @Transaction()
  async InitLedger (ctx: Context) {
    // initialize the ledger with activity data

    for await (const password of samplePasswords) {
      try {
        await this.CreatePassword(ctx, password, password.UserId);
      } catch (err) {
        console.log(err);
      }
    }
  }

  CorrectQueryString (queryString: string | QueryString): string {
    const parsedQueryString = typeof queryString === 'string'
      ? unmarshal(queryString) as QueryString
      : queryString;
    const correctedQueryString = stringify({
      ...parsedQueryString,
      selector: {
        ...parsedQueryString.selector,
        docType: 'password'
      }
    });
  };
  return correctedQueryString;
}

CreatePasswordKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('password', [id]);
}

async ReadPassword (ctx: Context, id: string): Promise<
  Uint8Array> {
  return readAsset(ctx, this.CreatePasswordKey(ctx, id));
}

async PasswordExists (ctx: Context, id: string): Promise<boolean>
  > {
  return assetExists(ctx, this.CreatePasswordKey(ctx, id));
}

import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from '
  fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination

```

```

  getLocation, getUser
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, Index, QueryString
} from './helpers/general.helper';

import { Location } from './models/location';
import {
  Pig, DetailedPig
} from './models/pig';
import { User } from './models/user';

import samplePigs from './samples/pig';

export class PigContract extends Contract {
  constructor () {
    super('org.porkwatch.pig');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /^d+$/ .test(param))
      ;
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate
        details do not match!');
  }

  async unknownTransaction (ctx: Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().
      fcn;
    throw new Error('Unknown transaction function: ${
      transactionName}');
  }

  @Transaction()
  @Param('state', 'Pig', 'Part formed JSON of Pig')
  async CreatePig (ctx: Context, state: Pig, transactorId: string)
    : Promise<string> {
    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Id !== state.RegisteredById)
      throw new Error('The user with Id: ${transactor.Id} is not
        the one who is registering the Pig');

    const exists = await this.PigExists(ctx, state.Id);
    if (exists)
      throw new Error('The Pig with Id: ${state.Id} already exists
        ');

    const createdPig = Pig.newInstance(state);
    const createdPigBytes = marshal(createdPig);

    const pigKey = this.CreatePigKey(ctx, createdPig.Id);
    await ctx.stub.putState(pigKey, createdPigBytes);

    const indexes: Index[] = [
      {
        name: 'pig_breed',
        fields: [createdPig.Breed]
      },
      {
        name: 'pig_birthDate',
        fields: [createdPig.BirthDate]
      },
      {
        name: 'pig_isMale',
        fields: [createdPig.IsMale.toString()]
      }
    ]
  }
}

```

```

];

for (let i = 0; i < indexes.length; i++) {
  const fields = indexes[i].fields.map((field) => field.
    toString() || 'null');
  const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
    fields);
  // Save index entry to state. Only the key name is needed, no
  // need to store a duplicate copy of the marble.
  // Note - passing a 'nil' value will effectively delete the
  // key from state, therefore we pass null character as
  // value
  await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
}

const eventPayload = marshal({
  ...createdPig,
  key: pigKey
});
ctx.stub.setEvent('CreatePig', eventPayload);

return stringify(createdPig);
}

@Transaction()
@Param('state', 'string', 'Part formed JSON of Pig')
async CreatePigWithStringState (ctx: Context, state: string,
  transactorId: string): Promise<string> {
  let returnedPig = '';
  try {
    const pig = JSON.parse(state) as Pig;
    returnedPig = await this.CreatePig(ctx, pig, transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedPig;
}

@Transaction()
@Param('state', 'Pig', 'Part formed JSON of Pig')
async UpdatePig (ctx: Context, state: Pig, transactorId: string)
  : Promise<string> {
  await retrieveUser(ctx, transactorId, true) as User;

  if (!state.Id)
    throw new Error('No pig Id provided');

  const existingPig = unmarshal(await this.ReadPig(ctx, state.Id
  )) as Pig;

  // 4th optional param here is an object that would
  // override the changes from state being copied into
  // existingPig
  // for updatedPig
  const updatedState = Object.assign({}, existingPig, state);
  const updatedPig = Pig.newInstance(updatedState);
  const updatedPigBytes = marshal(updatedPig);

  // no need to do anything with the CouchDB indexes for update
  // transactions

  const pigKey = this.CreatePigKey(ctx, updatedPig.Id);
  await ctx.stub.putState(pigKey, updatedPigBytes);

  const eventPayload = marshal({
    ...updatedPig,
    key: pigKey
  });
  ctx.stub.setEvent('UpdatePig', eventPayload);

  return stringify(updatedPig);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of Pig')
async UpdatePigWithStringState (ctx: Context, state: string,
  transactorId: string): Promise<string> {
  let returnedPig = '';
  try {
    const pig = JSON.parse(state) as Pig;
    returnedPig = await this.UpdatePig(ctx, pig, transactorId);
  } catch (err) {
    console.log(err);
  }
  return returnedPig;
}

@Transaction(false)
@Returns('string')
async GetPigHistory (ctx: Context, id: string): Promise<string>
  {
  const results = await getAssetHistory(ctx, this.CreatePigKey(
    ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'pig'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultForQueryString(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllPigs (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'pig'
    }
  });
  return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetPig (ctx: Context, id: string, isDetailed?:boolean):
  Promise<string> {
  const pig = unmarshal(await this.ReadPig(ctx, id)) as Pig;
  if (!pig || !Object.keys(pig))
    throw new Error('The Pig with Id: ${id} does not exist');

  if (!isDetailed)
    return stringify(pig);

  const mother = pig.MotherId ? unmarshal(await this.ReadPig(ctx
    , pig.MotherId)) as Pig : null;
  const father = pig.FatherId ? unmarshal(await this.ReadPig(ctx
    , pig.FatherId)) as Pig : null;
  let children: Pig[] = [];
  for (const id of pig.ChildrenIds) {
    const child = unmarshal(await this.ReadPig(ctx, id)) as Pig;
    children.push(child);
  }

  let locations: Location[] = [];
  for (const locationHistory of pig.LocationHistory) {
    const location = unmarshal(await getLocation(ctx,
      locationHistory.LocationId)) as Location;
    locations.push(location);
  }

  const registeredBy = unmarshal(await getUser(ctx, pig.
    RegisteredById)) as User;

  const detailedPig: DetailedPig = {
    ...pig,
    Mother: mother,
    Father: father,
    Children: children,
    Locations: locations,
    RegisteredBy: registeredBy
  }

  return stringify(detailedPig);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data
  for await (const pig of samplePigs) {

```



```

    try {
      await this.CreatePig(ctx, pig, pig.RegisteredById);
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('PigInitLedger', Buffer.from('Pig Ledger
    Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'pig'
    }
  });
  return correctedQueryString;
}

CreatePigKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('pig', [id]);
}

async ReadPig (ctx: Context, id: string): Promise<Uint8Array> {
  return readAsset(ctx, this.CreatePigKey(ctx, id));
}

async PigExists (ctx: Context, id: string): Promise<boolean> {
  return assetExists(ctx, this.CreatePigKey(ctx, id));
}

import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from '
  fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination
  ,
  getLocation, getPig, getUser
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Location } from './models/location';
import { Pig } from './models/pig';
import { Product, DetailedProduct } from './models/product';
import { User } from './models/user';

import sampleProducts from './samples/product';

export class ProductContract extends Contract {

  constructor () {
    super('org.porkwatch.product');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactionId = params.find((param) => /\d+$/ .test(param))
      ;
    if (!transactionId)
      throw new Error('No transactionId found, transactionId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');
  }

  const transactionor = await retrieveUser(ctx, transactionId, true)
    as User;
  if (transactionor.Email !== getClientCommonName(ctx))
    throw new Error('User credentials and X.509 certificate
      details do not match!');
}

async unknownTransaction (ctx:Context): Promise<void> {
  const transactionName = ctx.stub.getFunctionAndParameters().
    fcn;
  throw new Error('Unknown transaction function: ${
    transactionName}');
}

@Transaction()
@Param('state', 'Product', 'Part formed JSON of Product')
async CreateProduct (ctx: Context, state: Product, transactionId:
  string): Promise<string> {
  const transactionor = await retrieveUser(ctx, transactionId, true)
    as User;
  if (transactionor.Id !== state.RegisteredById)
    throw new Error('The User with Id: ${transactionor.Id} is not
      the one who registered the Product');

  const exists = await this.ProductExists(ctx, state.Id);
  if (exists)
    throw new Error('The Product with Id: ${state.Id} already
      exists');

  const createdProduct = Product.newInstance(state);
  const createdProductBytes = marshal(createdProduct);

  const productKey = this.CreateProductKey(ctx, createdProduct.
    Id);
  await ctx.stub.putState(productKey, createdProductBytes);

  const indexes = [
    {
      name: 'product_pigId',
      fields: [createdProduct.PigId]
    },
    {
      name: 'product_slaughterhouseId',
      fields: [createdProduct.SlaughterhouseId]
    },
    {
      name: 'product_retailerLocationId',
      fields: [createdProduct.RetailerLocationId]
    }
  ];

  for (let i = 0; i < indexes.length; i++) {
    const fields = indexes[i].fields.map((field) => field.
      toString() || 'null');
    const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
      fields);
    // Save index entry to state. Only the key name is needed, no
    // need to store a duplicate copy of the marble.
    // Note - passing a 'nil' value will effectively delete the
    // key from state, therefore we pass null character as
    // value
    await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
  }

  const eventPayload = marshal({
    ...createdProduct,
    key: productKey
  });
  ctx.stub.setEvent('CreateProduct', eventPayload);

  return stringify(createdProduct);
}

@Transaction()
@Param('state', 'string', 'Part formed JSON of Product')
async CreateProductWithStringState (ctx: Context, state: string,
  transactionId: string): Promise<string> {
  let returnedProduct = '';
  try {
    const product = JSON.parse(state) as Product;
    returnedProduct = await this.CreateProduct(ctx, product,
      transactionId);
  } catch (err) {
    console.log(err);
  }
  return returnedProduct;
}

@Transaction()
@Param('state', 'Product', 'Part formed JSON of Product')
async UpdateProduct (ctx: Context, state: Product, transactionId:

```

```

        string): Promise<string> {
    await retrieveUser(ctx, transactorId, true) as User;

    if (!state.Id)
        throw new Error('No product Id provided');

    const existingProduct = unmarshal(await this.ReadProduct(ctx,
        state.Id)) as Product;

    // 4th optional param here is an object that would
    // override the changes from state being copied into
    // existingProduct
    // for updatedProduct
    const updatedState = Object.assign({}, existingProduct, state)
    ;
    const updatedProduct = Product.newInstance(updatedState);
    const updatedProductBytes = marshal(updatedProduct);

    // no need to do anything with the CouchDB indexes for update
    // transactions

    const productKey = this.CreateProductKey(ctx, updatedProduct.
        Id);
    await ctx.stub.putState(productKey, updatedProductBytes);

    const eventPayload = marshal({
        ...updatedProduct,
        key: productKey
    });
    ctx.stub.setEvent('UpdateProduct', eventPayload);

    return stringify(updatedProduct);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of
    Product')
async UpdateProductWithStringState (ctx: Context, state: string,
    transactorId: string): Promise<string> {
    let returnedProduct = '';
    try {
        const product = JSON.parse(state) as Product;
        returnedProduct = await this.UpdateProduct(ctx, product,
            transactorId);
    } catch (err) {
        console.log(err);
    }
    return returnedProduct;
}

@Transaction(false)
@Returns('string')
async GetProductHistory (ctx: Context, id: string): Promise<
    string> {
    const results = await getAssetHistory(ctx, this.
        CreateProductKey(ctx, id));
    return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
    const totalCount = unmarshal(await getCount(ctx, 'product'));
    return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
    string): Promise<string> {
    const correctedQueryString = this.CorrectQueryString(
        queryString);
    const results = unmarshal(await getQueryResultForQueryString(
        ctx, correctedQueryString)) as AssetJsonRes[];
    return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
    string): Promise<string> {
    const correctedQueryString = this.CorrectQueryString(
        queryString);
    const results = unmarshal(await getQueryResultWithPagination(
        ctx, correctedQueryString)) as AssetJsonRes[];
    return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllProducts (ctx: Context): Promise<string> {
    const queryString = stringify({
        selector: {
            docType: 'product'
        }
    });
    return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetProduct (ctx: Context, id: string, isDetailed?:boolean)
    : Promise<string> {
    const product = unmarshal(await this.ReadProduct(ctx, id) as
        Product;
    if (!product || !Object.keys(product))
        throw new Error('The Product with Id: ${id} does not exist');

    if (!isDetailed)
        return stringify(product);

    const pig = unmarshal(await getPig(ctx, product.PigId) as Pig
        ;
    const registeredBy = unmarshal(await getUser(ctx, product.
        RegisteredById) as User;
    const slaughterhouse = unmarshal(await getLocation(ctx,
        product.SlaughterhouseId) as Location;
    const retailer = product.RetailerId
        ? unmarshal(await getUser(ctx, product.RetailerId) as User
            : null;
    const retailerLocation = product.RetailerLocationId
        ? unmarshal(await getLocation(ctx, product.RetailerLocationId
            )) as Location
            : null;

    const detailedProduct: DetailedProduct = {
        ...product,
        Pig: pig,
        RegisteredBy: registeredBy,
        Slaughterhouse: slaughterhouse,
        Retailer: retailer,
        RetailerLocation: retailerLocation
    };

    return stringify(detailedProduct);
}

@Transaction()
async InitLedger (ctx: Context) {
    // initialize the ledger with activity data

    for await (const product of sampleProducts) {
        try {
            await this.CreateProduct(ctx, product, product.
                RegisteredById);
        } catch (err) {
            console.log(err);
        }
    }

    ctx.stub.setEvent('ProductInitLedger', Buffer.from('Product
        Ledger Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
    const parsedQueryString = typeof queryString === 'string'
        ? unmarshal(queryString) as QueryString
        : queryString;
    const correctedQueryString = stringify({
        ...parsedQueryString,
        selector: {
            ...parsedQueryString.selector,
            docType: 'product'
        }
    });
    return correctedQueryString;
}

CreateProductKey (ctx: Context, id: string): string {
    return ctx.stub.createCompositeKey('product', [id]);
}

async ReadProduct (ctx: Context, id: string): Promise<Uint8Array
    > {
    return readAsset(ctx, this.CreateProductKey(ctx, id));
}

async ProductExists (ctx: Context, id: string): Promise<boolean>
    {
    return assetExists(ctx, this.CreateProductKey(ctx, id));
}
}

```

```

import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from '
fabric-contract-api';

import {
  assetExists,
  marshal, unmarshal,
  getClientCommonName, isTransactionSubmitting,
  readAsset, retrieveUser, getAssetHistory, getCount,
  getQueryResultForQueryString, getQueryResultWithPagination
  ,
  getAuction, getBid, getBuyOrder, getLocation, getPig, getProduct
  , getUser
} from './helpers/chaincode.helper';
import {
  AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Auction } from './models/auction';
import { Bid } from './models/bid';
import { BuyOrder } from './models/buyOrder';
import { Location } from './models/location';
import { Transfer, DetailedTransfer } from './models/transfer';
import { Pig } from './models/pig';
import { Product } from './models/product';
import { User } from './models/user';

import sampleTransfers from './samples/transfer';

export class TransferContract extends Contract {

  constructor () {
    super('org.porkwatch.transfer');
  }

  async beforeTransaction (ctx: Context): Promise<void> {
    const funcAndParams = ctx.stub.getFunctionAndParameters();
    const transactionName = funcAndParams.fcn;

    console.log('transactionName', transactionName);
    console.log('isTransactionSubmitting', isTransactionSubmitting
      (transactionName));
    console.log('clientCommonName', getClientCommonName(ctx));
    if (transactionName.endsWith('InitLedger')) {
      // Skip custom logic for InitLedger transaction
      return;
    }

    const params = funcAndParams.params;
    if (!isTransactionSubmitting(transactionName))
      return;

    let transactorId = params.find((param) => /^d+$/ .test(param))
      ;
    if (!transactorId)
      throw new Error('No transactorId found, transactorId, e.g.
        req.user.Id, is necessary for all transactions
        updating the ledger');

    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if (transactor.Email !== getClientCommonName(ctx))
      throw new Error('User credentials and X.509 certificate
        details do not match');
  }

  async unknownTransaction (ctx:Context): Promise<void> {
    const transactionName = ctx.stub.getFunctionAndParameters().
      fcn;
    throw new Error('Unknown transaction function: ${
      transactionName}');
  }

  @Transaction()
  @Param('state', 'Transfer', 'Part formed JSON of Transfer')
  async CreateTransfer (ctx: Context, state: Transfer,
    transactorId: string): Promise<string> {
    const transactor = await retrieveUser(ctx, transactorId, true)
      as User;
    if ((!state.ProductIds || !state.ProductIds.length) &&
      transactor.LocationId !== state.TransferFromId)
      throw new Error('The user with Id: ${transactor.Id} is not
        from the seller of the Transfer');
    else if ((state.ProductIds && state.ProductIds.length) &&
      transactor.LocationId !== state.TransferToId)
      throw new Error('The user with Id: ${transactor.Id} is not
        from the buyer of the Transfer');

    const exists = await this.TransferExists(ctx, state.Id);
    if (exists)
      throw new Error('The Transfer with Id: ${state.Id} already
        exists');

    const createdTransfer = Transfer.newInstance(state);
    const createdTransferBytes = marshal(createdTransfer);

    const transferKey = this.CreateTransferKey(ctx,
      createdTransfer.Id);
    await ctx.stub.putState(transferKey, createdTransferBytes);

    const indexes = [
      {
        name: 'transfer_transferFromId',
        fields: [createdTransfer.TransferFromId]
      },
      {
        name: 'transfer_transferToId',
        fields: [createdTransfer.TransferToId]
      },
      {
        name: 'transfer_startDate',
        fields: [createdTransfer.StartDate]
      },
      {
        name: 'transfer_transferDate',
        fields: [createdTransfer.TransferDate]
      },
      {
        name: 'transfer_acceptedDate',
        fields: [createdTransfer.AcceptedDate]
      }
    ];

    for (let i = 0; i < indexes.length; i++) {
      const fields = indexes[i].fields.map((field) => field.
        toString() || 'null');
      const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
        fields);
      // Save index entry to state. Only the key name is needed, no
        need to store a duplicate copy of the marble.
      // Note - passing a 'nil' value will effectively delete the
        key from state, therefore we pass null character as
        value
      await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
    }

    const eventPayload = marshal({
      ...createdTransfer,
      key: transferKey
    });
    ctx.stub.setEvent('CreateTransfer', eventPayload);

    return stringify(createdTransfer);
  }

  @Transaction()
  @Param('state', 'string', 'Part formed JSON of Transfer')
  async CreateTransferWithStringState (ctx: Context, state: string
    , transactorId: string): Promise<string> {
    let returnedTransfer = '';
    try {
      const transfer = JSON.parse(state) as Transfer;
      returnedTransfer = await this.CreateTransfer(ctx, transfer,
        transactorId);
    } catch (err) {
      console.log(err);
    }
    return returnedTransfer;
  }

  @Transaction()
  @Param('state', 'Transfer', 'Part formed JSON of Transfer')
  async UpdateTransfer (ctx: Context, state: Transfer,
    transactorId: string): Promise<string> {
    await retrieveUser(ctx, transactorId, true) as User;

    if (!state.Id)
      throw new Error('No transfer Id provided');

    const existingTransfer = unmarshal(await this.ReadTransfer(ctx
      , state.Id)) as Transfer;

    // 4th optional param here is an object that would
    // override the changes from state being copied into
    existingTransfer
    // for updatedTransfer
    const updatedState = Object.assign({}, existingTransfer, state
      );
    const updatedTransfer = Transfer.newInstance(updatedState);
    const updatedTransferBytes = marshal(updatedTransfer);

    // no need to do anything with the CouchDB indexes for update
    transactions

```

```

const transferKey = this.CreateTransferKey(ctx,
  updatedTransfer.Id);
await ctx.stub.putState(transferKey, updatedTransferBytes);

const eventPayload = marshal({
  ...updatedTransfer,
  key: transferKey
});
ctx.stub.setEvent('UpdateTransfer', eventPayload);

return stringify(updatedTransfer);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of
  Transfer')
async UpdateTransferWithStringState (ctx: Context, state: string
  , transferId: string): Promise<string> {
  let returnedTransfer = '';
  try {
    const transfer = JSON.parse(state) as Transfer;
    returnedTransfer = await this.UpdateTransfer(ctx, transfer,
      transferId);
  } catch (err) {
    console.log(err);
  }
  return returnedTransfer;
}

@Transaction(false)
@Returns('string')
async GetTransferHistory (ctx: Context, id: string): Promise<
  string> {
  const results = await getAssetHistory(ctx, this.
    CreateTransferKey(ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'transfer'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultForQueryString(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllTransfers (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'transfer'
    }
  });
  return this.GetQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetTransfer (ctx: Context, id: string, isDetailed?:boolean
  ): Promise<string> {
  const transfer = unmarshal(await this.ReadTransfer(ctx, id))
    as Transfer;
  if (!transfer || !Object.keys(transfer))
    throw new Error('The Transfer with Id: ${id} does not exist')
    ;
  if (!isDetailed)
    return stringify(transfer);
}

const auction = transfer.AuctionId
  ? unmarshal(await getAuction(ctx, transfer.AuctionId)) as
    Auction
  : null;
const bid = transfer.BidId
  ? unmarshal(await getBid(ctx, transfer.BidId)) as Bid
  : null;
const buyOrder = transfer.BuyOrderId
  ? unmarshal(await getBuyOrder(ctx, transfer.BuyOrderId)) as
    BuyOrder
  : null;
const pig = unmarshal(await getPig(ctx, transfer.PigId)) as
  Pig;
let products: Product[] = [];
for (const id of transfer.ProductIds) {
  const product = unmarshal(await getProduct(ctx, id)) as
    Product;
  products.push(product);
}
const seller = unmarshal(await getUser(ctx, transfer.SellerId)
  ) as User;
const buyer = unmarshal(await getUser(ctx, transfer.BuyerId))
  as User;
const transferFrom = unmarshal(await getLocation(ctx, transfer
  .TransferFromId)) as Location;
const transferTo = unmarshal(await getLocation(ctx, transfer.
  TransferToId)) as Location;
const canceledBySeller = transfer.CanceledBySellerId
  ? unmarshal(await getUser(ctx, transfer.CanceledBySellerId))
    as User
  : null;
const canceledByBuyer = transfer.CanceledByBuyerId
  ? unmarshal(await getUser(ctx, transfer.CanceledByBuyerId))
    as User
  : null;
const acceptedBySeller = transfer.AcceptedBySellerId
  ? unmarshal(await getUser(ctx, transfer.AcceptedBySellerId))
    as User
  : null;
const acceptedByBuyer = transfer.AcceptedByBuyerId
  ? unmarshal(await getUser(ctx, transfer.AcceptedByBuyerId))
    as User
  : null;

const detailedTransfer: DetailedTransfer = {
  ...transfer,
  Auction: auction,
  Bid: bid,
  BuyOrder: buyOrder,
  Pig: pig,
  Products: products,
  Seller: seller,
  Buyer: buyer,
  TransferFrom: transferFrom,
  TransferTo: transferTo,
  CanceledBySeller: canceledBySeller,
  CanceledByBuyer: canceledByBuyer,
  AcceptedBySeller: acceptedBySeller,
  AcceptedByBuyer: acceptedByBuyer
};

return stringify(detailedTransfer);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data

  for await (const transfer of sampleTransfers) {
    try {
      await this.CreateTransfer(ctx, transfer, transfer.SellerId)
      ;
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('TransferInitLedger', Buffer.from('Transfer
    Ledger Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'transfer'
    }
  });
};

```

```

    return correctedQueryString;
}

CreateTransferKey (ctx: Context, id: string): string {
    return ctx.stub.createCompositeKey('transfer', [id]);
}

async ReadTransfer (ctx: Context, id: string): Promise<
    Uint8Array> {
    return readAsset(ctx, this.CreateTransferKey(ctx, id));
}

async TransferExists (ctx: Context, id: string): Promise<boolean
    > {
    return assetExists(ctx, this.CreateTransferKey(ctx, id));
}

import stringify from 'json-stringify-deterministic';

import { Context, Contract, Param, Returns, Transaction } from '
    fabric-contract-api';

import {
    assetExists,
    marshal, unmarshal,
    getClientCommonName, isTransactionSubmitting,
    readAsset, retrieveUser, getAssetHistory, getCount,
    getQueryResultForQueryString, getQueryResultWithPagination
    ,
    getLocation
} from './helpers/chaincode.helper';
import {
    AssetJsonRes, QueryString
} from './helpers/general.helper';

import { Location } from './models/location';
import { User, DetailedUser } from './models/user';

import sampleUsers from './samples/user';

export class UserContract extends Contract {

    constructor () {
        super('org.porkwatch.user');
    }

    async beforeTransaction (ctx: Context): Promise<void> {
        const funcAndParams = ctx.stub.getFunctionAndParameters();
        const transactionName = funcAndParams.fcn;

        console.log('transactionName', transactionName);
        console.log('isTransactionSubmitting', isTransactionSubmitting
            (transactionName));
        console.log('clientCommonName', getClientCommonName(ctx));
        if (transactionName.endsWith('InitLedger')) {
            // Skip custom logic for InitLedger transaction
            return;
        }

        const params = funcAndParams.params;
        if (!isTransactionSubmitting(transactionName))
            return;

        let transactorId = params.find((param) => /^d+$/ .test(param))
            ;
        if (!transactorId)
            throw new Error('No transactorId found, transactorId, e.g.
                req.user.Id, is necessary for all transactions
                updating the ledger');

        const transactor = await retrieveUser(ctx, transactorId, true)
            as User;
        if (transactor.Email !== getClientCommonName(ctx))
            throw new Error('User credentials and X.509 certificate
                details do not match!');
    }

    async unknownTransaction (ctx:Context): Promise<void> {
        const transactionName = ctx.stub.getFunctionAndParameters().
            fcn;
        throw new Error('Unknown transaction function: ${
            transactionName}');
    }

    @Transaction()
    @Param('state', 'User', 'Part formed JSON of User')
    async CreateUser (ctx: Context, state: User, transactorId:
        string): Promise<string> {
        const transactor = await retrieveUser(ctx, transactorId) as
            User;

        if (transactorId !== state.Id && transactor.Id !== state.
            RegisteredById)
            throw new Error('The User with Id: ${transactor.Id} is not
                the one who is registering the User');

        const exists = await this.UserExists(ctx, state.Id);
        if (exists)
            throw new Error('The User with Id: ${state.Id} already exists
                ');

        const createdUser = User.newInstance(state);
        const createdUserBytes = marshal(createdUser);

        const userKey = this.CreateUserKey(ctx, createdUser.Id);
        await ctx.stub.putState(userKey, createdUserBytes);

        const indexes = [
            {
                name: 'user_locationId',
                fields: [createdUser.LocationId]
            },
            {
                name: 'user_role',
                fields: [createdUser.Role]
            },
            {
                name: 'user_birthDate',
                fields: [createdUser.BirthDate]
            },
            {
                name: 'user_registrationDate',
                fields: [createdUser.RegistrationDate]
            }
        ];

        for (let i = 0; i < indexes.length; i++) {
            const fields = indexes[i].fields.map((field) => field.
                toString() || 'null');
            const indexKey = ctx.stub.createCompositeKey(indexes[i].name,
                fields);
            // Save index entry to state. Only the key name is needed, no
            // need to store a duplicate copy of the marble.
            // Note - passing a 'nil' value will effectively delete the
            // key from state, therefore we pass null character as
            // value
            await ctx.stub.putState(indexKey, Buffer.from('\u0000'));
        }

        const eventPayload = marshal({
            ...createdUser,
            key: userKey
        });
        ctx.stub.setEvent('CreateUser', eventPayload);

        return stringify(createdUser);
    }

    @Transaction()
    @Param('state', 'string', 'Part formed JSON of User')
    async CreateUserWithStringState (ctx: Context, state: string,
        transactorId: string): Promise<string> {
        let returnedUser = '';
        try {
            const user = JSON.parse(state) as User;
            returnedUser = await this.CreateUser(ctx, user, transactorId)
                ;
        } catch (err) {
            console.log(err);
        }
        return returnedUser;
    }

    @Transaction()
    @Param('state', 'User', 'Part formed JSON of User')
    async UpdateUser (ctx: Context, state: User, transactorId:
        string): Promise<string> {
        await retrieveUser(ctx, transactorId, true) as User;

        if (!state.Id)
            throw new Error('No user Id provided');

        const existingUser = unmarshal(await this.ReadUser(ctx, state.
            Id)) as User;

        // 4th optional param here is an object that would
        // override the changes from state being copied into
        // existingUser
        // for updatedUser
        const updatedState = Object.assign({}, existingUser, state);
        const updatedUser = User.newInstance(updatedState);
        const updatedUserBytes = marshal(updatedUser);

```

```

// no need to do anything with the CouchDB indexes for update
// transactions

const userKey = this.CreateUserKey(ctx, updatedUser.Id);
await ctx.stub.putState(userKey, updatedUserBytes);

const eventPayload = marshal({
  ...updatedUser,
  key: userKey
});
ctx.stub.setEvent('UpdateUser', eventPayload);

return stringify(updatedUser);
}

@Transaction()
@Param('state', 'string', 'Stringified part formed JSON of User
')
async UpdateUserWithStringState (ctx: Context, state: string,
  transactorId: string): Promise<string> {
  let returnedUser = '';
  try {
    const user = JSON.parse(state) as User;
    returnedUser = await this.UpdateUser(ctx, user, transactorId)
    ;
  } catch (err) {
    console.log(err);
  }
  return returnedUser;
}

@Transaction(false)
@Returns('string')
async GetUserHistory (ctx: Context, id: string): Promise<string>
{
  const results = await getAssetHistory(ctx, this.CreateUserKey(
    ctx, id));
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetCount (ctx: Context): Promise<string> {
  const totalCount = unmarshal(await getCount(ctx, 'user'));
  return stringify(totalCount);
}

@Transaction(false)
@Returns('string')
async GetQueryResultForQueryString (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultForQueryString(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetQueryResultWithPagination (ctx: Context, queryString:
  string): Promise<string> {
  const correctedQueryString = this.CorrectQueryString(
    queryString);
  const results = unmarshal(await getQueryResultWithPagination(
    ctx, correctedQueryString)) as AssetJsonRes[];
  return stringify(results);
}

@Transaction(false)
@Returns('string')
async GetAllUsers (ctx: Context): Promise<string> {
  const queryString = stringify({
    selector: {
      docType: 'user'
    }
  });
  return await getQueryResultForQueryString(ctx, queryString);
}

@Transaction(false)
@Returns('string')
async GetUser (ctx: Context, id: string, isDetailed?:boolean):
  Promise<string> {
  const user = unmarshal(await this.ReadUser(ctx, id)) as User;
  if (!user || !Object.keys(user))
    throw new Error('The User with Id: ${id} does not exist');

  if (!isDetailed)
    return stringify(user);

  const location = unmarshal(await getLocation(ctx, user.
    LocationId)) as Location;
  const registeredBy = unmarshal(await this.ReadUser(ctx, user.
    RegisteredById)) as User;

  const detailedUser: DetailedUser = {
    ...user,
    Location: location,
    RegisteredBy: registeredBy
  };

  return stringify(detailedUser);
}

@Transaction()
async InitLedger (ctx: Context) {
  // initialize the ledger with activity data

  for await (const user of sampleUsers) {
    try {
      await this.CreateUser(ctx, user, user.RegisteredById);
    } catch (err) {
      console.log(err);
    }
  }

  ctx.stub.setEvent('UserInitLedger', Buffer.from('User Ledger
    Initialized'));
}

CorrectQueryString (queryString: string | QueryString): string {
  const parsedQueryString = typeof queryString === 'string'
    ? unmarshal(queryString) as QueryString
    : queryString;
  const correctedQueryString = stringify({
    ...parsedQueryString,
    selector: {
      ...parsedQueryString.selector,
      docType: 'user'
    }
  });
  return correctedQueryString;
}

CreateUserKey (ctx: Context, id: string): string {
  return ctx.stub.createCompositeKey('user', [id]);
}

async ReadUser (ctx: Context, id: string): Promise<Uint8Array> {
  return readAsset(ctx, this.CreateUserKey(ctx, id));
}

async UserExists (ctx: Context, id: string): Promise<boolean> {
  return assetExists(ctx, this.CreateUserKey(ctx, id));
}

```

XI. Acknowledgment

First and foremost, I would like to thank Sir Marbert Marasigan and Sir Richard Bryann Chua for their continued guidance in the year it took to complete this SP. The system implemented in this study made use of many technologies I had no prior experience of, and their insight greatly helped with both writing the manuscript and the implementation of the system.

I would also like to thank Sir Alberto D. Burdeos, former Asst. Dept. Head III of the Veterinary Inspection Board of the City of Manila, for answering my inquiries into the country's pig industry. His experience formed many of the schemas I used for the assets in the system.

I would also like to thank the discord community of the Hyperledger Foundation for answering my inquiries into Hyperledger Fabric, its documentation, and for giving me some sample repositories to learn from. The blockchain system of this paper based much of its infrastructure from the sample repository with the name of full-stack-asset-transfer-guide. Setting up a blockchain network for production would not have been possible in the time I was given without the sample infrastructure and guide from this repository, and I would like to thank all the developers who worked on it.