4th September 2020

University of Leicester
Department of Informatics

CO7201 Individual Project
Final Report

Development of Smart Contract Blockchain Applications in DAML

author
Alex Maragakis

am1146@student.le.ac.uk
199039301

supervisor
Prof. Reiko Heckel

second marker
Prof. Yudong Zhang

Total Word Count	18,042
Minus Appendices	14,557

Abstract

In recent years, smart contract blockchain applications have become a topic of 'hype' in the software development community. The technology promises benefits in trust and privacy by removing centralised authorities and relying heavily on encryption to protect data integrity. Being based in the realm of cryptography and distributed systems, the development of these dApps (*decentralised applications*) using low-level development tools requires a fairly sophisticated knowledge of complex computer science fields.

Fortunately, some technologies have been created that make development accessible to developers that are not so versed in these specific disciplines. One such technology is DAML; a language provided by the company Digital Asset. DAML is a 'low code' smart contracting language that hides blockchain implementation from the developer to allow them to focus on the smart contract model itself. The language is fairly new and does not have a large user base in the programming community.

This project centres around the development of a smart contract blockchain application in DAML. The application has a React/TypeScript user interface. It is developed based on requirements inspired by the real-world business processes of a third-party company. The benefits and drawbacks of the language are evaluated in the process of development. This report documents the requirements of the application, the development process, and the benefits and drawbacks of the language. There is also a discussion that brings to question the necessity of using blockchain, as opposed to a traditional centralised database, in the application.

Declaration

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Alex Maragakis
4th September 2020

Confidentiality

The functional requirements for the application developed in this project were inspired by requirements outlined through iterative discussion with a real company. The app requirements and underlying business model were shared with myself and another development team in confidence. In order to prevent a breach of confidence, some steps have been taken to conceal the identity of the company and the exact nature of the business model. In this report, I shall simply refer to the company as *The Company*.

I make efforts in this report to present user stories in terms that do not expose the underlying business model. This is made easier by the fact that my application features have in fact diverged from the disclosed requirements from *The Company*. I have intentionally altered the presentation of some business processes, where possible, both in the report and application.

My application will not be used by *The Company* and cannot be considered a prototype. Another student development team has developed a prototype application according to the specific requirements, and their application is not discussed here.

Contents

1. Introduction

The Company has expressed an interest to develop a system that tracks products in an agricultural supply chain. The product records would be immutable, and created by users without the intervention from a trusted centralised authority. Records should only be disclosed to product owners and parties that are currently handling the product. It has been proposed that a *blockchain* could provide a suitable implementation paradigm for the storing of these product records.

The Company has paired with another team of students from the University to create a prototype product. They, *The Company* and the development team, have kindly included me in discussions about requirements and business processes. They have given permission for me to use their real-world application requirements as inspiration for my own project; *Development of Smart Contract Blockchain Applications in DAML*.

This academic project is an *implementation/technical* project that centres around the development of a smart contract blockchain application written in DAML.

The main aim of the project is to assess DAML as a practical development tool for smart contract developers. DAML is a relatively new language, and does not have a large online user community, if compared to tools provided on the Ethereum platform. DAML's simplicity makes it a good option for non-specialised developers to transition to blockchain from a traditional web background. Put simply, DAML has the potential to be a valuable language in the future of blockchain; this project aims to test it out.

Development of an application based on real-world requirements should provide a suitable basis for assessing DAML. The main *objective* of the project is to implement this application with a solely DAML-based blockchain backend. The DAML backend should successfully communicate with a frontend written using popular web technologies.

The result of the development effort is an application which can successfully present an audit history of product records, all stored on a DAML ledger, to users in a React user interface. The DAML backend application has been tested using DAML's own testing framework. The combined front and backend system has been assessed through user testing. The system is initialised with sample data demonstrating a complete agricultural product process; from raw harvest to sale.

2. Background

2.1. Blockchain

The modern *blockchain* (BC) emerged in 2008 with the proposal of Bitcoin. It was formally introduced in a paper entitled ‘Bitcoin: A Peer-to-Peer Electronic Cash System’ [1]. The author outlined a novel system that allows digital ‘cash’ to be securely transferred in a way that avoids the need for centralised authorities, such as banks. The exclusion of trusted authorities makes Bitcoin *trustless*.

The system stores transaction data in a chain of data blocks. Each block contains the cryptographic hash of the previous. This is with the intention that blocks cannot be arbitrarily modified without failing chained hash reference checks. Modifying any individual block requires that all subsequent blocks need to be updated to hold correct hash references of previous blocks. Recomputing the hashes for the entire chain, in order to conceal the alteration of a given block, is made infeasible through *proof-of-work* (PoW) criteria. In the Bitcoin system, computed hashes need to adhere to strict conditions; for example, by starting with some predefined prefix characters. This restriction means that the vast majority of computed hashes are considered invalid. Hash algorithms need to be run multiple times on the data block, each iteration with a varied *nonce* field on the data block, in order to arrive at an acceptable hash. The inability to alter the contents of already-stored data blocks makes blockchain data *immutable*.

Any ‘incorrect’ information stored on the chain in an intermediate block would have to be compensated for in a new block at the end of the chain. For example, ‘mistakenly’ sending 1 Bitcoin to someone can only be rectified by putting a new block on the end of the chain where the recipient sends 1 Bitcoin back to the original sender. The original transaction cannot be altered or deleted.

The transaction data is kept on a publicly available *ledger* that is locally storable on user machines. The globally recognised state of the ledger is determined through a consensus mechanism that considers the majority state of all participating locally stored ledgers. Maintaining a ledger in this distributed manner is the basis of *Distributed Ledger Technologies* (DLTs). In the Bitcoin case, transactions may hold information about Bitcoin transferrals between users. It is possible to store arbitrary information formats on blockchains, hence the emergence of usage examples from varied application domains.

Blockchain has proven to be attractive to developers. According to a recent survey by Deloitte [2], 55% of participating software organisations said that blockchain would be ‘critical, in our top five strategic priorities’ with top use cases including ‘digital currency’, ‘data access and sharing’ and ‘data reconciliation’. According to Blockchain Council [3], top use cases for blockchain include ‘supply chain management’ and ‘digital identity’.

It is argued by some that this surge in popularity is little more than hype. Some high profile commentators refer to this broad adoption as ‘chainwashing’ [4]. *Chainwashing* is the practice of employing blockchain in applications that do not benefit from it. The success and novelty of the Bitcoin system may have led developers and business leaders to naively see DLT as a silver bullet.

2.2. Smart Contracts

Smart Contacts (SCs) were first proposed by computer scientist and legal scholar Nick Szabo in his 1997 theoretical paper ‘Formalizing and Securing Relationships on Public Networks’ [5]. In the publication, Szabo claims that the traditional, physical contract is ‘the basic building block of a market economy’. He proposes a computerised contract implementation with the promise of ‘[reducing] mental and computational transaction costs’. Like a traditional contract, smart contracts are agreements between parties that allow relationships to be formalised. Unlike a traditional contract, smart contracts can automatically execute actions when agreed conditions are met. Szabo asserted that cryptographic solutions would play an important role in maintaining contract integrity, and in preventing contract information being disclosed to non-stakeholder parties.

In his 1998 essay ‘Secure Property Titles with Owner Authority’ [6], he elaborates on a use case of smart contracts for tracking property rights. A possible implementation is proposed that uses public key cryptography to secure the property’s current title and the ‘chain’ of all previous titles. The article hints at the use of some form of distributed ledger, or ‘replicated database technology’, for maintaining the title information in a consensus based manner.

In the 2013 ‘Ethereum Whitepaper’ [7], Ethereum founder Vitalik Buterin claimed that the scripting available in the Bitcoin protocol can ‘facilitate a weak version of a concept of “smart contracts”’. Buterin goes on to propose a blockchain platform, the Ethereum platform, that provides ‘a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create “contracts” that can be used to encode arbitrary state transition functions’. The Ethereum platform supports the creation of smart contract-based *decentralised applications* (dApps) within domains that extend beyond cryptocurrency. Like Bitcoin, the Ethereum platform is a DLT that uses a consensus mechanism to maintain ledger state in a trustless manner.

The general understanding of modern smart contracts within the software development community is that they are not necessarily associated with the creation of formal, binding relationships, as is the case with traditional contracts. Rather they are seen as ‘automatically executable lines of code that are stored on a blockchain’ [8].

Smart contracts can be implemented in many languages. Some languages have been designed specifically for use with smart contracts, such as Simplicity, DAML, and Ethereum’s Solidity. General purpose languages such as C++ can also be used to implement blockchains and smart contracts, but an endeavour with a non-specialised language would require knowledge of low-level blockchain implementation.

2.3. DAML

Implementing a distributed blockchain ledger from the ground up requires knowledge of some specialised computer science fields, including but not limited to distributed systems and cryptography. High-level blockchain implementation languages that run on existing platforms will allow non-specialised software developers to adopt blockchain with reduced difficulty.

The *Digital Asset Modelling Language* (DAML) is a smart contract implementation language created by the company *Digital Asset* (DA). DAML is a high-level language that hides underlying blockchain implementation details with the intention of allowing the developer to focus on the functional requirements of the smart contract application. DA claim that DAML is so high-level, abstract, and syntactically simple that its comprehension is feasible for non-programmers [9].

DAML is currently a *relatively* new language, and is not yet widely adopted. At the time of writing this, the DAML YouTube channel only has 230 subscribers [10], and most of the online articles you will find written about DAML are authored by DA employees.

DAML has been used in some fairly large scale FinTech projects. DA has been contracted by the Australian Stock Exchange (ASX) to redevelop their antiquated CHESS trading system in DAML [11]. DAML is also being used by Accenture to develop full-stack software license management applications [12].

2.3.1. DAML Crash-Course

An example DAML file is provided in the code appendix; *ServiceRequest.daml*.

DAML is a functional language and is syntactically similar to Haskell. In DAML code, smart contracts are defined as *templates.* Contract templates contain many terms that may appear on traditional contracts. DAML contracts define stakeholder roles; *signatories*, *controllers* and *observers*.

2.3.1.1. Signatories

Signatories are the parties that provide authorisation on contracts. In order for a contract to be created, the listed stakeholders need to have taken some action in its creation. In the `ServiceRequest` contract, the `signatory` is the `client` Party. This means that `client` 'must consent to the creation of an instance of this contract' [13], and that `client` is 'the [party] who would be put into an *obligable position* when this contract is created'. In the `ServiceOrder` contract there are two signatories, `client` and `serviceProvider`. If `client` and `serviceProvider` are two distinct parties, a `ServiceOrder` can only be created through a mechanism that allows both parties to provide consent. This means that the `ServiceOrder` cannot be created 'from-scratch' (like the single-signatory `ServiceRequest`), but needs to be created through a *choice* mechanism (see *Controllers and Choices*, below).

2.3.1.2 Controllers and Choices

Controllers are parties that can execute *choices* (actions) associated with the given contract. The `controller` on the `ServiceRequest` contract is `serviceProvider`. `serviceProvider` could choose to *exercise* either an `Accept` or a `Decline` choice. If he/she decides to exercise an `Accept`, a `ServiceOrder` is created. The `ServiceOrder` requires consent from both `client` and `serviceProvider`. Since `client` provided consent on the `ServiceRequest` contract, which defines `Accept`, and `serviceProvider` executed `Accept`, it is considered that both parties have provided consent to create a `ServiceOrder`, and hence it is created. Exercising a choice causes the contract that defined the choice to become *archived*. This means that the `ServiceRequest` contract can no longer be acted upon, though it still exists on the ledger in memory. Executing the `Decline` choice would simply cause the `ServiceRequest` to be archived, whilst `Accept` would archive the `ServiceRequest` and create a `ServiceOrder`.

It is within *choices* that the on-chain logic associated with smart contract blockchains occurs. The `Accept` choice here is rather simplistic; it merely creates an instance of a `ServiceOrder` contract using fields already specified in the `ServiceRequest` and one additional choice argument, `time`, that is passed when the choice is exercised. Choices can contain *assertions*; checks that ensure specific conditions are met, lest the choice execution be aborted. Choice code can perform *some* rudimentary calculations; DAML is not a Turing-complete language [14].

The return type of a choice is specified after the choice name. Choices should typically return `ContractIds` of the contracts that the choice creates. The `Decline` choice does not create a new contract, and hence does not return a `ContractId`.

2.3.1.3. Observers

Observers are parties that do not provide authorisation, and do not have the ability to exercise choices, but are able to view the data stored on the contract. In `ServiceRequest` and `ServiceOrder`, the `observer` is `auditor`. `auditor` is able to fetch the contract to review its contents, but is not able to take any action beyond that. Content on the DAML ledger is encrypted in such a way that non-observing parties cannot view the contract. All parties that are listed as either signatories, controllers, or observers, are considered observing parties.

3. Related Work

3.1. Blockchain-based Supply Chains

It is claimed that one of the most promising use cases for blockchain is in supply chain management. The immutability of blockchain data makes it difficult to retrospectively alter records to present a false product history. According to Deloitte, using blockchain in supply chains can 'increase traceability of material supply chain to ensure corporate standards are met' [15], and '[decrease] losses from counterfeit/grey-market trading'.

3.1.1. Provenance

In 2016, Provenance, a UK Startup, used blockchain technology to track a tuna fish caught in Indonesia 'from landing to factory and beyond' [16]. By using a blockchain to track internationally traded food, they aim to 'eradicate fraudulent reporting', and hence to 'incentivise *ethical labour* practice and *environmental preservation*'. The idea is that suppliers cannot falsely claim to have ethically sourced produce without insufficient, or fraudulent, reporting.

An example of how a lack of visibility can enable unethical practices can be seen in the case of CP Foods. CP Foods, a prawn supplier, that supplies food to many major UK and US supermarkets, including Tesco, has been linked to slavery on Thai prawn shipping vessels [17]. When queried about the problem, CP Food's UK managing director said 'We know there's issues with regard to the [raw] material that comes in [to port], but to what extent that is, we just don't have the visibility'.

On the environmental side, claims that produce is sustainably sourced would similarly need to be backed up in reporting. Case-in-point, sustainably sourced tuna can only be caught in certain parts of the ocean, and with certain fishing practices (such as pole-and-line). A tuna is sustainably sourced in the eyes of a UK supermarket if the reporting history behind the tuna indicates that the tuna was caught in a sustainable manner.

The benefits of the Provenance system, if instantiated at all points in the supply chain, seem to be many. According to Provenance, 'The data is stored in an immutable, decentralised, globally-auditable format which protects identities by default, allowing for secure data verification' [16]. The immutability of the database means that data cannot be retrospectively altered to hide unethical practices. The decentralised nature of the data indicates that there are no trusted authorities that have explicit power over the data stored. The globally-auditable format allows any party to view the data stored on the blockchain to verify claims made by suppliers.

The Provenance blockchain is a public blockchain. This means that there are no 'administrator' entities that influence the chain's consensus mechanism. It also means that anyone can sign up to the network for the purposes of creating or viewing data. Public blockchains are said to be more computationally expensive to run, and more difficult to manage.

3.1.2. Innover Digital

Recently, Innover Digital, a software consulting company that intends to '[transform] Service and Supply value-chains' [18], created a prototype distributed application in DAML for procuring Covid-19 health supplies [19]. The aim of the project was to create a supply chain application that could provide the 'ability for every player to review the demand and supply patterns'. It was developed in partnership with Digital Asset employees.

The resulting prototype tracked healthcare products from manufacturers through to suppliers, shippers, buyers, customs agencies and ultimately hospitals. It including a mechanism for requesting quotes, and allowed potential buyers to view available products from manually on-boarded sellers. Their application not only tracked records, but had a mechanism for making requests to service providers, all of which could be rejected or accepted with results recorded on-chain using DAML *choices*.

The developer from Innover Digital that worked on the project said that DAML had presented many benefits. DAML's contract visibility model made it easy to keep product quotes private between specific buyers and sellers. He stated that the entire backend of the business process for the application was developed in '3 days. …a testament to how powerful DAML is'. He also suggested that the fact that 'the DAML ledger exposes easy to REST APIs', for querying the ledger using traditional front end web technologies, 'did shave many months off development time'. In fact, the most time consuming part of the application development came in developing the UI frontend, taking 'several weeks to get right'.

The experience of Innover Digital is a ringing endorsement of DAML. Their choice to use DAML had been based on their indecision of underlying blockchain platform. DAML is portable to various different blockchain platforms, including HyperLedger Fabric and Amazon Quantum Ledger [20], without changes to code. DAML is also interoperable between other DAML applications running on different underlying blockchains.

3.2. Blockchain Development Methodologies

Having first existed in the world as Bitcoin, blockchain is just over a decade old. This makes it a relatively juvenile implementation paradigm. This has not prevented its large-scale adopted. As a result of its youth, there are no established methods and processes specific to blockchain development [21]. It could be argued that there is no need for specialised methods, and that existing software development practices, such as those commonly found in Agile, also transfer to blockchain. Nonetheless, some practices have been proposed that could aid efforts to develop smart contract blockchain applications.

According to Marchesi, Marchesi, and Tonelli [22], 'the first step to develop a software system using sound software engineering practices is to have a clear development process'. In their paper, they propose a method that employs various techniques found in Agile software, such as user stories, and producing data models in the form of *Unified Modelling Language* (UML) diagrams.

The method of Marchesi et al. is an 8-step process; *partially* outlined here. It starts with the provision of a simple 'goal of the system' statement to developers. Later, developers are expected to write 'system requirements in terms of user stories'. The on-chain and off-chain split should then be defined. The actors and actions present in the smart contracts should then be specified and implemented. Once the smart contracts are defined, the user interface is designed and implemented. Finally, the system is tested and deployed.

As well as their 8-step method, Marchesi's paper proposes a method of modelling smart contracts using UML class diagrams. UML diagrams are first derived from the user stories. They are then modified to consider the underlying smart contract implementation technology such that they are easily transferable to smart contract code.

Rocha and Ducasse [21] also propose the use of UML as one of three possible approaches to modelling blockchain oriented software. As well as UML, they suggest the use of Entity Relationship Models (ER models) and Business Process Model and Notation (BPMN) to model smart contracts.

In the proposal of Rocha et al., each modelling approach covers a different aspect of the smart contract design process. ER models and UML would typically be used to represent how data fields in smart contracts are related to each other, as in the case of Marchesi et al.'s proposal. BPMN diagrams would then be used to represent the workflow of the smart contract application.

Seebacher and Maleshkova [23] provide a model-driven methodology for describing blockchain business networks that use Blockchain Business Network Ontologies (BBOs). These ontologies aim to capture all information that could be used to describe the network. They have the flexibility to capture participant roles, contract actions and data, and descriptors of network architecture. Such ontologies could also be used to describe non-blockchain software. UML and its variations have a wider adoption in practice, despite the added flexibility available in ontology diagrams.

4. Project Requirements

Project requirements were arrived at through a process of regular discussion with *The Company*. This involved the iterative creation of user stories and data models. This is inline with the proposed Agile methodology of Marchesi [22]. As mentioned in the *Confidentiality* section, the description of some requirements have been altered in this academic project in order to conceal the exact underlying business model.

4.1. Overview

The application is an agricultural product tracking and trading application. Users use the application to record steps in a product's history. These steps include raw material production, processing, transportation, handover, and sales. The application allows product owners and handlers to view the entire history of their products.

The main technical requirement for this academic project is that a DAML blockchain is used to store product records. For the sake of assessing DAML as a full-stack development tool, it is also required that the application has a user interface implemented using a popular web technology.

4.2. User Stories

The functional requirements of this application were specified as *User Stories* (USs). User stories relating to features that were ultimately decided by *The Company* to be driven by off-chain backend tech stacks are not considered in this academic project. USs are roughly divided by user role (see *Design - Assigned User Roles*).

4.2.1 Producer User Story

“As a Producer, I want to create production records for raw produce that I have harvested. I want to record the type of product, the amount harvested, the plot from which it was harvested, the time of harvest, and a unique label associated with the plot-specific product batch”.

4.2.2. Processor User Story

“As a Processor, I want to create processing records for processing services that I have provided. I want to record information about the input product(s), including type(s), unique label(s), and amount(s). I also want to record the same fields of information for the output product. I want to record the start and end times of the process, and the name/location of the processing facility.”

4.2.3. Transporter User Story

“As a Transporter, I want to create transportation records for transportation services that I have provided. I want to record information about the product(s), including type(s), unique label(s) and amount(s). I want to record the start time and location, and the end time and location.”

4.2.4. Trader User Story

The Company had hinted at a sophisticated trading mechanism, but it was decided that this would mostly take place off-chain. Here is included a modified user story that has been added to this academic project.

“As a Trader, I want to make and receive offers on products in the system. I want to be able to view all the records of the product that has been offered to me. Once I have purchased, or sold, a product, I want to create a sale record. The sale record will hold information about the product, the sale price, information about the buyer and seller, and the time of sale.”

5. Design

After requirements were set, decisions were made that were not directly specified by *The Company* in the first instance. Unlike the user stories, these design decisions were initially proposed by either myself or the prototype-developing student development team. Some of these design decisions are unique to this academic project. For example, some in-app processes have been extended with unique features in order to extend the usage of DAML within the project.

5.1. User Roles

Each user plays a different role in the product supply chain. Some of these roles are constant, *assigned* to the user's account, whilst others are *implied*, changing with time depending on the user's current activities.

5.1.1. Implied User Roles

Every user in the application has the ability to trade and handover produce. This means that every type of user can potentially have the *implied role* of *Handler* or *Owner.*

A user cannot be given a new implied role without his/her consent. This is thanks to the request and approval process that is unique to the academic project and was not specified by *The Company* (see *Design - In-App Workflows*).

5.1.1.1. Handlers

All parties are potential product *handlers*. A handler has a product in their possession. It is possible for a user to have a product in their possession that they do not currently own. It is usually the case that service providers, such as processors and transporters, become product handlers.

5.1.1.2. Owners

All parties are potential product *owners*. A product owner owns the product in a legal sense. Owners have likely bought the product at some point in its history, or are producers of not-yet-traded products.

5.1.2 Assigned User Roles

Assigned Roles are unchanging roles that are associated to a user's account. The assigned roles are *Producer, Processor, Transporter, Trader* and *Admin.* Each of these role types represent players in an agricultural product supply chain. The UI restricts the abilities of each user to actions related to their role.

5.1.2.1. Producers

Producers are harvesters of raw produce. Producers are seen as product owners until they sell the product to someone else. They are also the initial handlers of the product until it is passed on to another party.

5.1.2.2. Processors

Processors process products. Given some input product(s), such as oranges, a processor applies a process and provides a processed output product, such as orange juice. During processing, the processor is considered the product's current handler.

5.1.2.3. Transporters

Transporters provide product transportation services. During transportation, the transporter is considered the product's current handler.

5.1.2.4. Traders

Traders are parties that are neither producers, processors or transporters, but can trade products. Since all parties can trade produce, Traders can be considered as supply chain players that don't directly alter the state of a product other than the state of its ownership.

5.1.2.5. Admins

Admins are parties that can do everything. They are mostly used for testing the application. They *could* be used in the real world to act on behalf of users that are struggling to use the technology.

5.2. Ledger Entities

Ledger Entities are pieces of information stored on the blockchain ledger.

5.2.1. Records

Records are uploaded by supply chain contributors to track product state. The history of a product can be seen as a series of steps represented by its records. For service records, i.e. processing and transportation records, *Start* and *End* records exist.

5.2.1.1. Production Records

Producers create *Production Records* to register the raw produce that they have contributed to the supply chain. Production Records specify the type of product, quantity of product, and an identifier to the plot from which the product was harvested. Additionally, Production Records specify a *product label*; a unique identifier that allows that individual product batch to be traced.

5.2.1.2. Processing Records

Processors create various types of *Processing Records* to allow stages of the processing process to be documented on the supply chain ledger. Processing Records specify information about the input product(s), information about the output product, start and end times, and process location. As in Production Records, product information includes the type of product, quantity of product, and product labels.

5.2.1.3. Transportation Records

Transportation Records are created by transporters to document transportation services. They contain product information, start and end times, and start and end locations.

5.2.1.4. Sale Records

When an item is sold, a *Sale Record* is put to the ledger. Sale Records contain information about the product, the identity of the buyer and seller, the price, and time of sale. In the real world, a given batch of agricultural could be sold at several stages; for example, first by an exporter/importer, and secondly by a supermarket.

5.2.1.5. Handover Records

When an item is handed over to another party, a *Handover Record* is created. Handover Records are similar Sale Records. They hold information about the identity of the new handler and the previous handler, information about the product, and the time of the handover.

5.2.2. Requests and Orders

Offers, handovers, and services are provided on a request basis. These requests are stored on the blockchain ledger. Unlike *Records*, they are not considered a part of a product's history. Requests for *services* result in Orders, that are also stored on the blockchain.

A unique request and order type exists for each Record type. This is with the exception of Production Records. This is because raw product harvest is not carried out by request.

Orders are divided into *New Orders* and *Active Orders.* New Orders exist for Products with an accepted Request and are the basis through which Start Records are created. Similarly, Active Orders exist for Products with started New Orders, and are used to create End Records.

Requests and orders are stored on the ledger because the contract actions available on these entities imply record creation on the ledger.

5.2.3. Products

In order to conveniently track records in DAML, a *Product* contract is created for each product batch. A Product ledger entity points to Records associated with a given product. For example, a raw product would point to one or more production records. A processed product would point to one or more production records and also a processing record. The Product contract also contains information about the type of product, current label(s), and accumulated amount, as well as the current owner and current handler.

It is through Product contracts that users exercise product update actions, such as adding production records. This was an independent design decision. The implication from discussions with *The Company* was that product history should be traced purely through tracing back the unique labels. Holding record references in a Product contract makes for much easier traceback, and allows updates to all records to be applied from a single choice on the Product contract.

5.2.4. Merges and Splits

To add flexibility to the Product type, Merges and Splits are permitted. When a product owner owns at least two of the same product, the owner can merge the two products. This process collates the unique labels and records and assigns them to the resulting merged product, with the amount set as the sum of the two original products. Splits allow owners to divide their product into two, choosing the amount going to each split. In the case of a split, the records and labels of the original product are shared between the two resulting products. Merges and Splits are recorded on the ledger. Like Records, their creation is derived from DAML choices that appear on the Product contract.

5.3. In-App Workflows

The in-app workflows are elaborations on the user stories as set out by *The Company*. Here, BPMNs are used to represent the workflow, as proposed by Rocha and Ducasse [21]. Many actions are intended to be restricted to specific user roles. In DAML alone, it is impossible to access controls on certain parties. As a result, the restrictions are enforced in the UI only.

In these workflow diagrams, *Update Product* refers to the process of *archiving* the existing Product contract and then creating a new one. Since the implementation paradigm is a blockchain, contract fields cannot be updated in-place. A red circle on the blockchain pool indicates an aborted transaction.

It is assumed in these diagrams that existing user products have already been fetched from the ledger.

5.3.1. New Products

Creating a new *Product* entity is an action available to Producers and Admins.

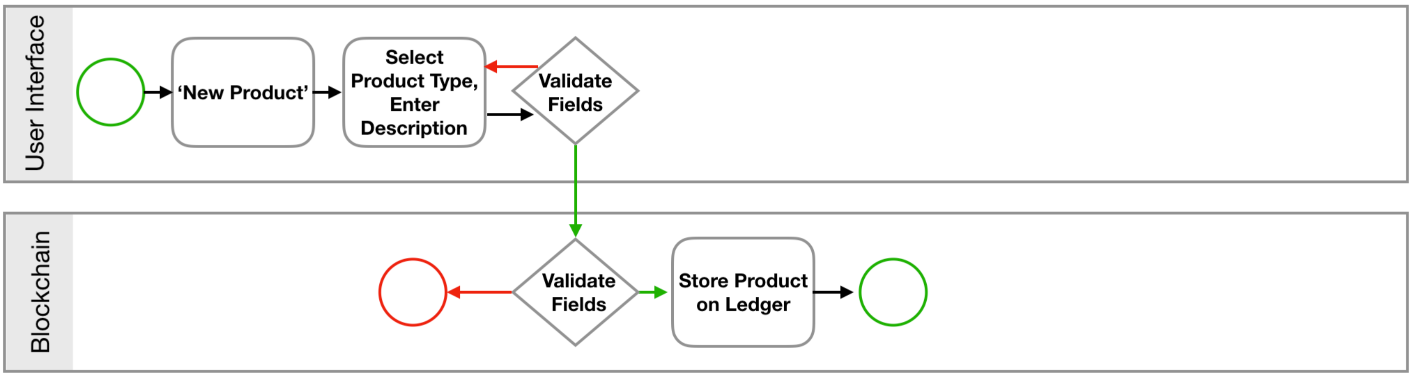

Figure 5.3.1: Create a New Product

Validation is performed both in the user interface, in order to provide immediate feedback on invalid fields, and also on the ledger, to avoid bad Products being made via the API. The type of product, *Product Type*, is selected from a drop down menu containing predefined product types; for example, bananas. The *Description* field allows the user to add a human readable descriptor to the batch in order to distinguish it from other product batches in their UI.

5.3.2. Production Records

Adding a *Production Record* to a Product is available to Producers and Admins. The user must be the current owner and handler of the product. The Product must be unprocessed and without any pending requests or orders.

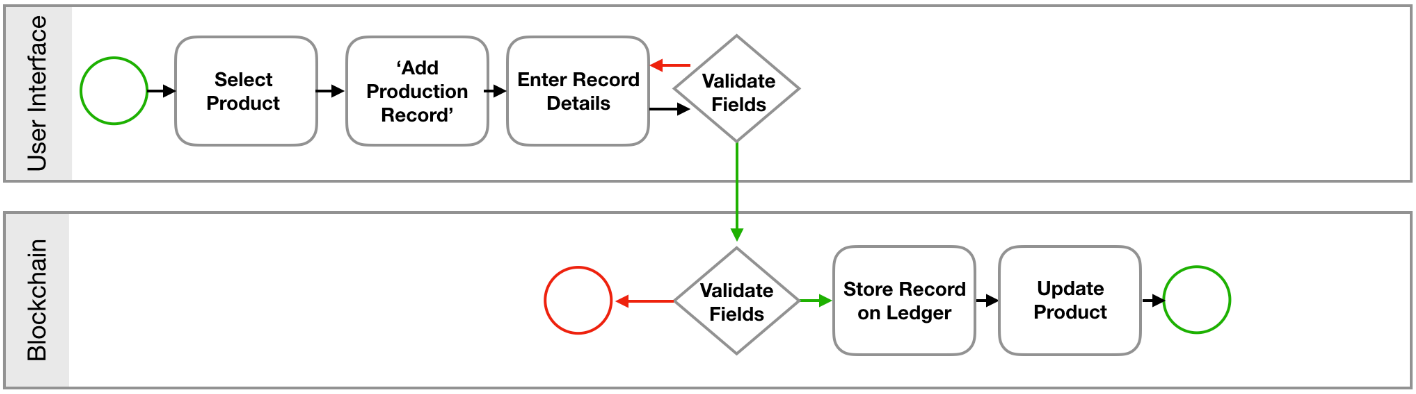

Figure 5.3.2: Add a Production Record

Previously created Products present an *'Add Production Record'* option. The Record details include: product label (autogenerated), amount (quantity and unit), plot (and farmer, through inference), location (name and country), and time of harvest. Product type is inferred from the Product on which the record is added.

5.3.3. Service Records

Service Records come in two forms; *Service Start Records* and *Service End Records*. The two available services in this application are processing and transportation. Hence, Processing Records and Transportation Records are Service Records

Service Start Records can be added in two ways. They can be added on a request basis (see *Start a Service Order)*, or they can be added immediately by service providers that own the product and are currently handling it (see *Add a Service Start Record Without Requests*).

Service End Records are added in roughly the same way for both immediate and request-based workflows. They simply require that the product is currently being serviced by them, and an *Active Service Order* exists for the product.

Though the BPMN examples are given for processing workflows, the workflow for transportation is equivalent.

5.3.3.1. Add a Service Start Record Without Requests

Adding Service Start Records without going through a request system can be done by service providers and admins that are the current owners and handlers of the product. The product must have no active requests or orders.

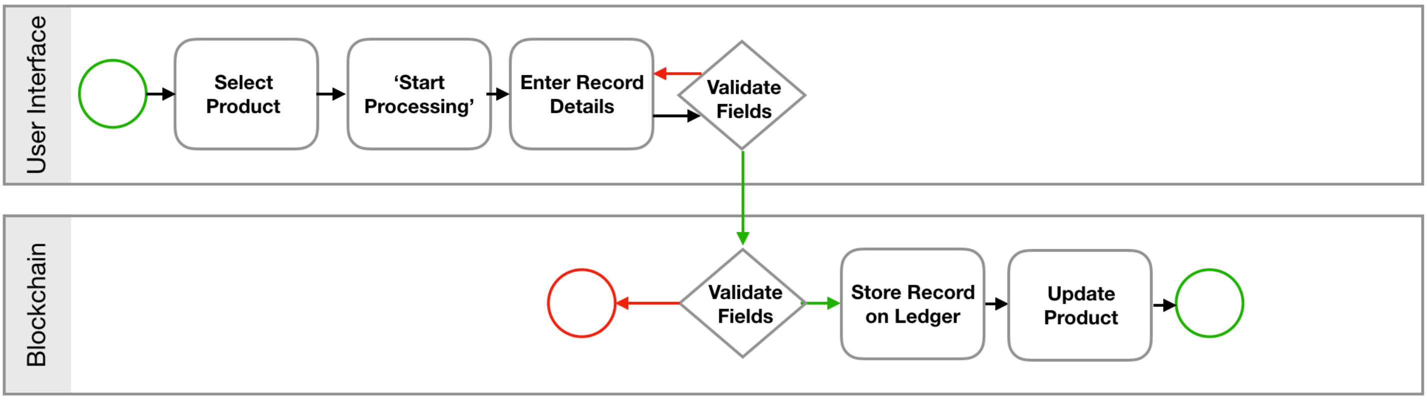

Figure 5.3.3.1: Add a Processing Start Record without requests

For processing services, record details include: process start time, desired output product, and process location. For transportation, they include: transportation start time, start location, and end location.

5.3.3.2. Make a Service Request

The request-based service workflow is available to all product owners. The product must have no active requests or orders. The recipient of the request must be a service provider.

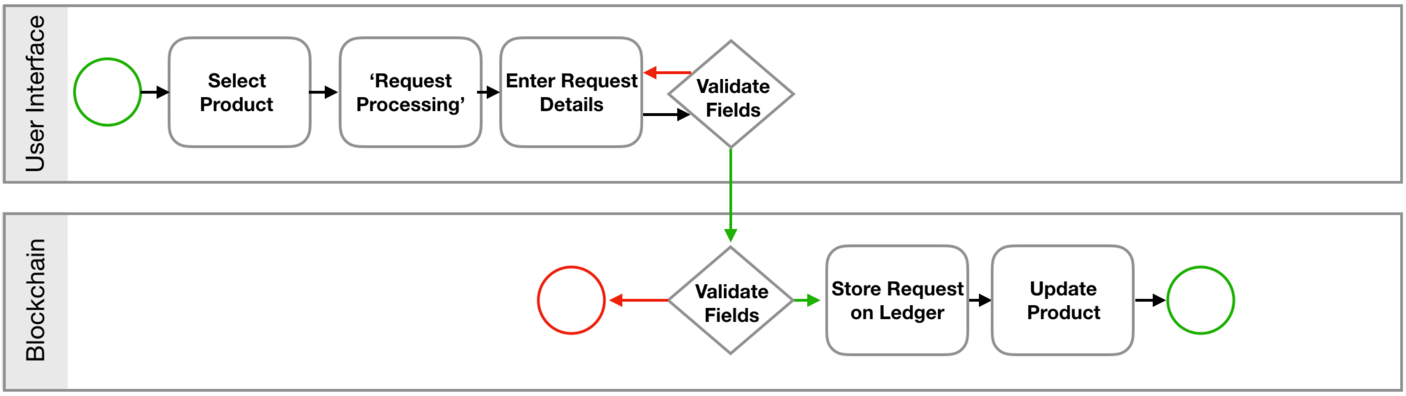

Figure 5.3.3.2: Make a Processing Request

Processing request details include desired output type, desired processor, process location (inferred from processor). Transportation request details include desired start and end locations, and desired transportation provider.

5.3.3.3. Responding to a Service Request

Recipients of Service Requests can choose to accept or decline them. Accepting the request implies a product handover, hence a Handover Record is stored. Additionally, a *New Service Order* is stored on the ledger.

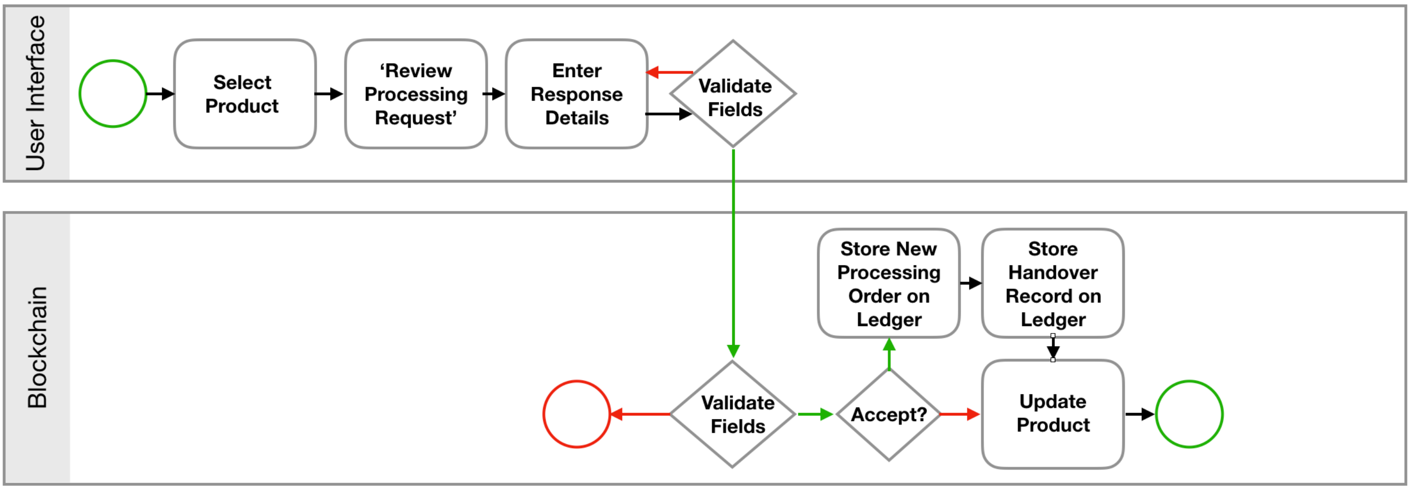

Figure 5.3.3.3: Responding to a Processing Request

The response details input dialog allows the user to accept or decline requests. If the service provider decides to accept the request, he/she is also required to enter the time of the product handover.

5.3.3.4 Start a Service Order

New Service Orders are created after requests are accepted. Service Start Records are created through the New Service Order when using a request-based workflow. Once an order has been initiated, an *Active Service Order* is created.

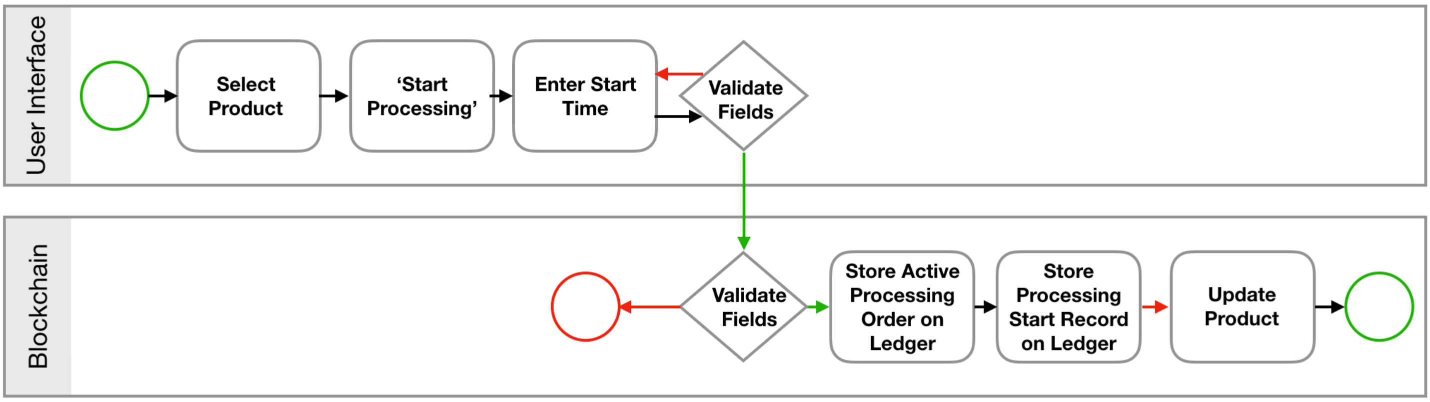

Figure 5.3.3.4: Start a Processing Order

The only field for validation is the start time. The start time should be later the time on the most recent record.

5.3.3.5 Complete a Service Order

Services are completed through the Active Service Order contract. The completion of the Active Service Order creates a *Service End Record.* It also returns the product to the idle state; the state it was in before the service was requested. The completion of a processing order updates the product type, label(s), and possibly the amount.

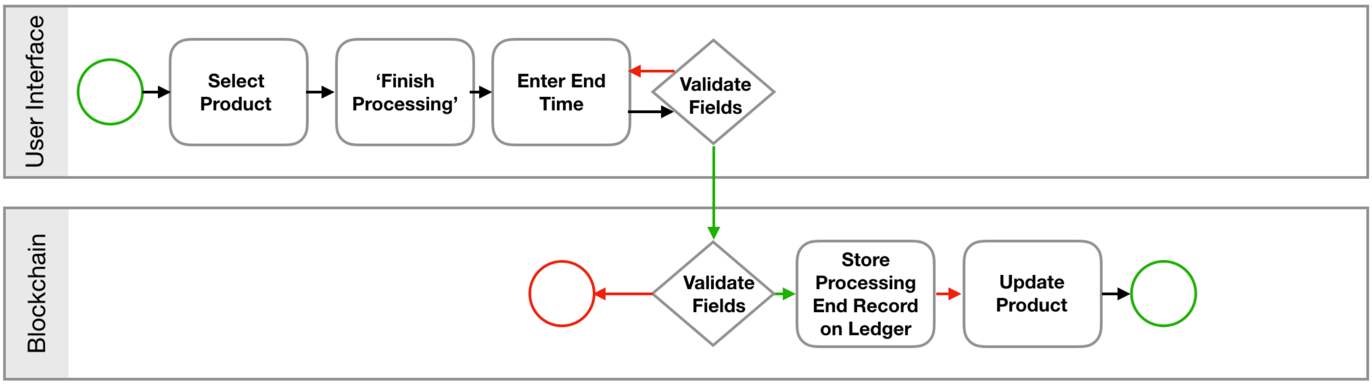

Figure 5.3.3.5: Complete a Processing Order

The only field for validation is the end time. As with the start time on the Service Start Record, the end time should be later than time on the most recent record.

5.3.4. Handover and Sale Records

The workflow for product handovers and sales is based on a request mechanism. This is to ensure that parties do not receive responsibility of products against their will. BPMN examples here depict the creation of Handover Records. The creation of Sale Records is a roughly equivalent process.

5.3.4.1 Making Handover and Sale Requests

Handover Requests are available to product handlers, whilst Sale Requests are available to product owners.

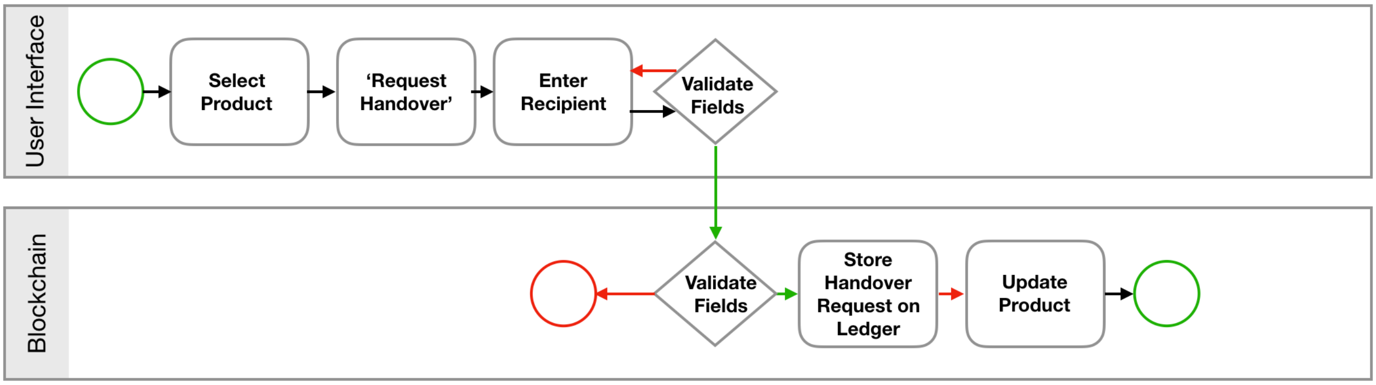

Figure 5.3.4.1 Make a Handover Requests

5.3.4.2 Responding to Handover and Sale Requests

If a handover request is accepted, the recipient becomes listed as the new handler of the Product. Similarly, sale request acceptance causes the recipient to become the new owner of the Product.

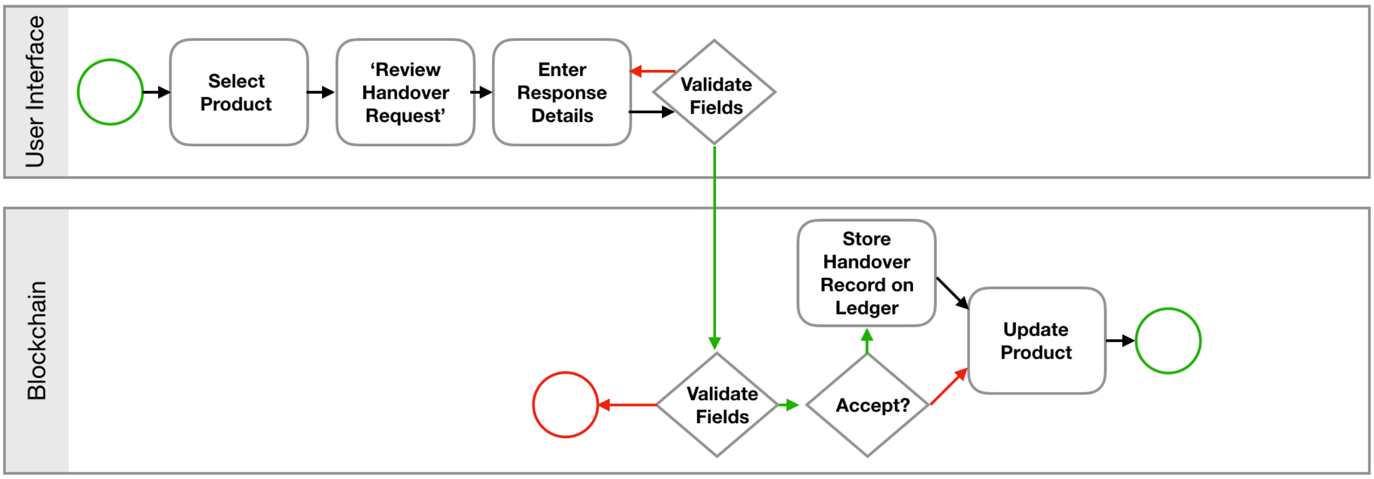

Figure 5.3.4.2 Responding to a Handover Request

Response details include accept/decline, and a time of transfer (in the case of accept).

5.3.5. Product Merges and Splits

Merging is the process of aggregating two products. In order to merge products, the user must be the handler and owner of both products, and both products must be of the same type. Splitting is the process of dividing an owned product into two. Like merging, it can only be performed on products that the user currently owns and handles. Merges *and* splits do not require a request process because the only stakeholder in the process is the product owner.

The BPMN represents the split process. The split process is equivalent, though with different details supplied in the form.

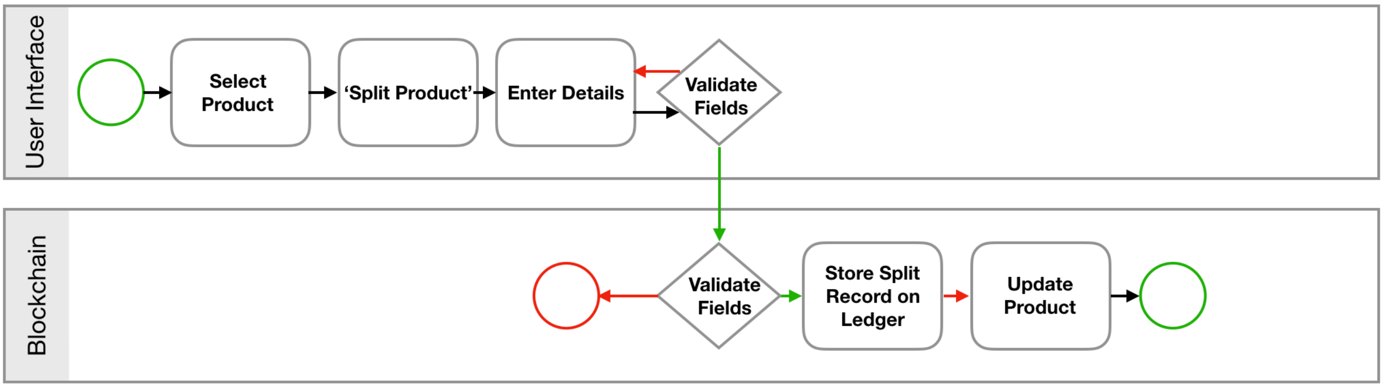

Figure 5.3.5: Splitting a product

In the case of a split, required details include split ratio, and new product descriptions for the resulting divisions. Merges require the specification of another product owned by the user, and a new description for the merged product.

5.4. Product State Transitions

Products can exist in various states, the default state being *IDLE.* Product states exist to prevent certain product actions being executed at times when the product is not allowed to be altered. For example, if a product is being processed, it cannot also be transported. The IDLE state is the most flexible, allowing the product to be altered in various ways.

The state transitions of the product can be represented as finite a state machine (*Figure 5.4*).

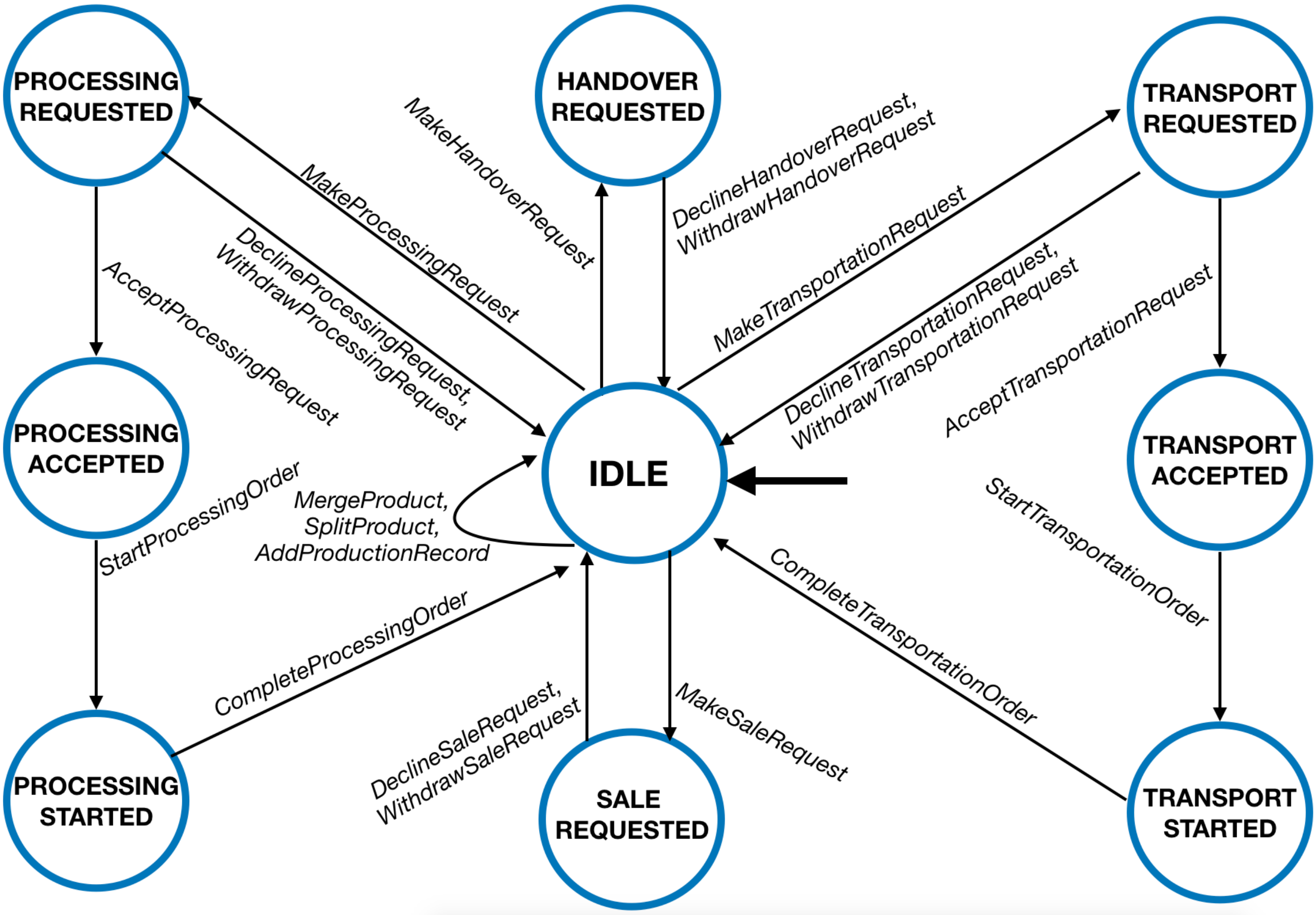

Figure 5.4: Product State Transitions

Transition inputs are named after DAML choices that exist on the Product contract, and on resulting request and order contracts.

5.5. Data Model

The on-chain data model is represented as UML (*Figure 5.5*), in accordance with the proposals of Rocha et al. [21], and Marchesi et al. [22].

Not all implementation classes are shown. For example, *RecordDetails* has an implementation for every kind of Record type, but only *ProductionRecordDetails* and *HandoverRecordDetails* are shown. Similarly, *OrderId* can be an identifier for any kind of order or request. Further details on the data model can be found in the project DAML files; they are virtually an exact mapping.

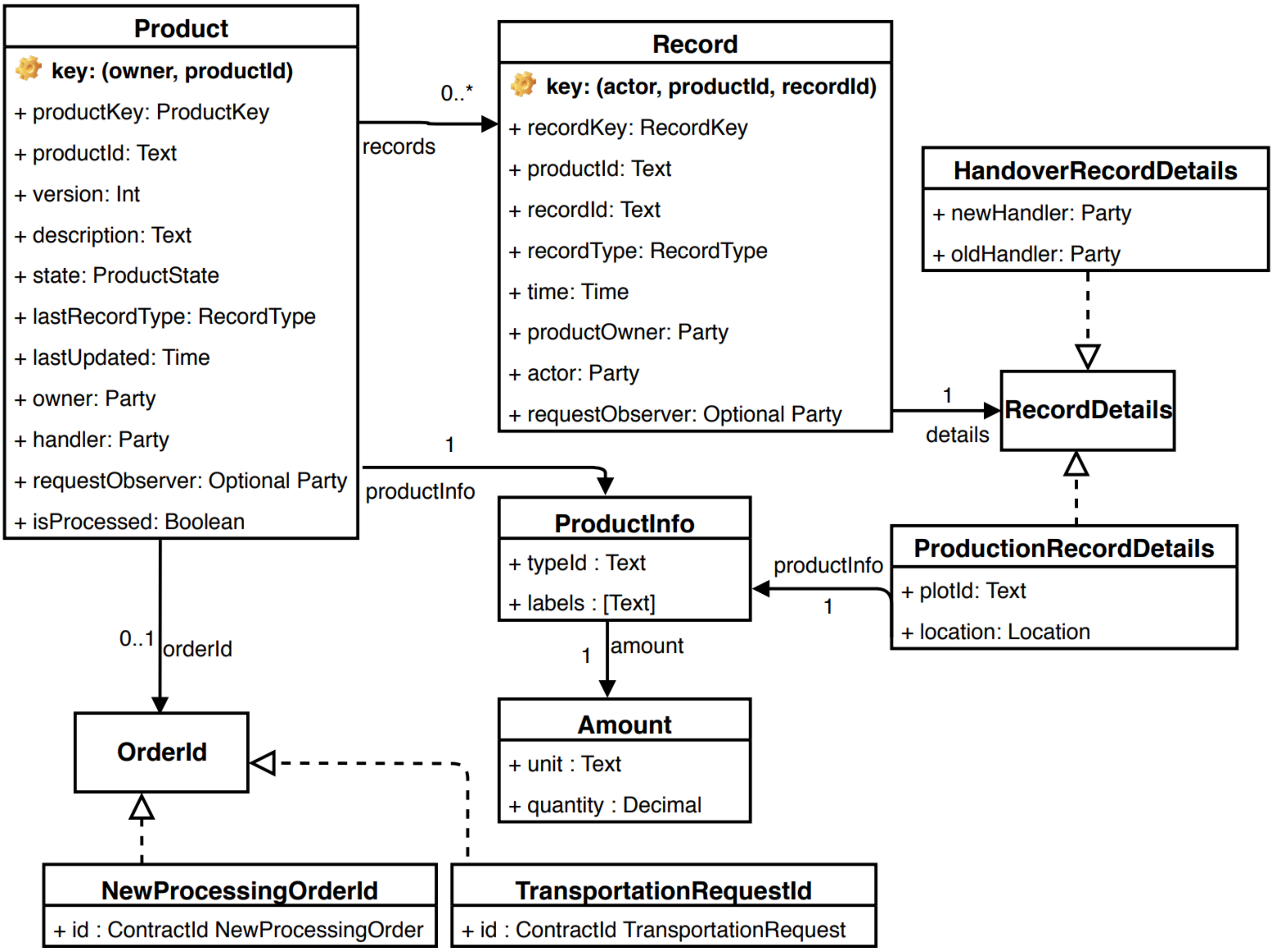

Figure 5.5: Data Model

5.5.1 Entity Identifiers

Products and Records each have a *Text* field that contains an identifier unique to the contract; *productId* and *recordId* respectively. In order to ensure uniqueness, *Globally Unique Identifiers* (GUIDs) can be used. These are generated on the client; DAML does not have a GUID generation facility.

Records are referenced from the Product as *RecordKeys*. In DAML, contracts can be uniquely referenced by a contract key. The benefit of contract keys, as opposed to contract identifiers, is that contract keys always reference the most recent version of the contract; a contract identifier becomes invalid if the contract is archived.

5.5.2 Record Implementation

All of the record types specified in the *Ledger Entities* section are implemented as *Record* contracts. Each Record contract contains a field, details, that holds a *RecordDetails* object. The RecordDetails is a union-typed object that can hold information about any type of Record; for example, *ProductionRecordDetails*, *ProcessingStartRecordDetails*, or *HandoverRecordDetails*. This implementation allows all records to be referenced from the *records* property (*recordKeys* in the DAML file) on the Product. It also makes for much easier contract fetching for the UI code.

5.5.3. Product Orders and Requests

Product order and request contracts are referenced from the Product in the *orderId* field. This is an optional field; empty if the Product is in the IDLE state. When not empty, orderId holds the contract identifier of the DAML contract representing the current order. For example, if a handover has been requested, the orderId field holds a *HandoverRequest* contract identifier.

It is safe to use contract identifiers, as opposed to contract keys, in this use case because all request and order contracts have a lifetime that extends for only one single DAML choice. This is because request and order contracts are considered to be expired as soon as any choice, e.g. *Accept*, is exercised on them. This is unlike the *Product* contract where choice execution should not cause the product itself to expire, even if the current *version* of the Product becomes archived.

The *requestObserver* field on the Product and Record allows recipients of requests to view the history of a product before accepting the request. The field is required for this functionality because request recipients may not be the product's owner or current handler, meaning that they are unable to observe the Product and its Records. The field is optional, and only has a value when the product is in a 'request' state; for example, the PROCESSING_REQUESTED state.

5.6. User Interface Design

Asides from a few rough ideas, the design of the user interface was mostly improvised during development. This section outlines the main presentational features of the user interface that were arrived at by the end of the development process.

5.6.1. Login Page

In order to present user-specific data and actions within the application, it was important to have a user login page. The design of the page is quite simple; a drop down user input and login button. In the application implementation, the available users are predefined in JS files.

5.6.2. Main Layout

The main page(s) of the application are all of similar layout.

There is a header bar that indicates the currently logged in user, and contains a signout button.

On the left is a sidebar that allows the user to select which type of ledger entities he/she would like to view. Each sidebar option contains the name of the page that it directs to, and also contains an icon that illustrates the page. For example, the *Production* button has a leaf icon.

In the centre of the screen is a table of records that varies depending on which sidebar tab is selected.

5.6.3. Products

The Products page displays a table that holds information on all the Product contracts that are visible to the current user. Each entry in the table can be selected.

When a product is selected, the *Product Audit* section is presented. The Product Audit section is built up of two main components; the *Product Audit Header* and the *Product Timeline*.

5.6.3.1. Product Audit Header

The Product Audit Header is entitled with the Product's description. Underneath this title is a small table holding non-record information about the current product; for example, the current owner and handler.

At the bottom of the header is a selection of user actions. When the product is in the IDLE state, the available actions are displayed as a row of buttons. Each button holds an icon corresponding to the action type, and are identical to the icons used in the sidebar. Clicking an icon produces a dropdown menu with available actions relating to the action group. When the product is not in the IDLE state, a single button is presented to the request recipient that allows them to act on the current request/order.

The header is a 'sticky' component that stays in the same position regardless of the vertical scroll position. This allows users to scroll down the product timeline and have actions visibly available to them constantly.

5.6.3.2. Product Editors

All editors allow users to specify fields for the product update. Invalid fields are highlighted red when the user clicks *Okay*. Otherwise, data is submitted to the ledger. All editors also have a *Cancel* button to exit the data entry process.

5.6.3.3. Product Timeline

The Product Timeline shows the record history of the current product. The timeline is a vertical series of record entries. Each record entry holds information about the record, and is attached to a central timeline connector by its corresponding icon and annotated by the time of the record.

5.6.4. Records

All the other sidebar options direct to Record pages. Each of these pages holds a table containing information about the records visible to the user. On pages for request-based records, the page contains a dropdown that allows the user to select which variant of record he/she would like to view. For example, on the *Handovers* page, the user can choose to view either Handover Requests or Handover Records.

6. Implementation

Implementation took approximately 7 weeks of full time work. The implementation of the current codebase began in full swing once requirements had been decided upon through discussion with *The Company*. Prior to this, the implementation was mostly experimental, and with the purpose of gaining familiarity with DAML. Here we discuss the implementation of the finished application.

The codebase is quite large so this section discusses only selected file examples. These examples are chosen as they best demonstrate the DAML workflow and/or do interesting things. The TypeScript/React files that are included make use of DAML client libraries.

The root directory of the codebase is at *am1146/code/trunk/app*. From there, the DAML codebase is in the *daml/* folder, and the UI code is in the *ui/src* folder.

6.1. DAML Implementation

The DAML files in the codebase saw many iterations. Initially, the *Product* contract did not exist, and the only templates where *ProductionRecord, ProcessingRecord,* and *TransportationRecord.* This reflected the raw requirements set out by the company, as reflected in the user stories. As mentioned in *Ledger Entities*, the DAML codebase ultimately expanded to include *Product*, and various request and order contracts.

6.1.1. Record Contract

6.1.1.1. Record Fields

`Record` is implemented in such a way that it is *like* a parent class for the various kinds of records. All records share certain fields, whilst some details vary. Class inheritance does not exist in DAML, so this is achieved by specifying a union-typed member field; `details`.

The type of the `details` field is `RecordDetails`. This is defined at line 30 in *Record.daml* (see *DAML Excerpts* in the Appendices). It is declared as a union type, where each union member is a *tag/value* pair. For example, one member of the union is `ProductionRecordDetailsTag ProductionRecordDetails`. This means that the `details` field will contain `ProductionRecordDetails` when set with the `ProductionRecordDetailsTag` tag. Extracting the type of record in DAML using the `details` tag can be slightly awkward. Hence, to easier identify the type of a record, the enum `RecordType` is defined (line 14) and used in the `recordType` field on the `Record`.

The `Record` contract can be uniquely identified on the ledger by means of a `RecordKey`. The `RecordKey` type is defined at line 9. It is a triple of `(Party, Text, Text)` with the intention of being used to store the `Record` fields `actor`, `productId`, and `recordId`. DAML keys must always have the contract's `maintainer` set as the first field. Here, the maintainer is the record creator and sole signatory: the `actor`. `productId` and `recordId` are GUIDs that uniquely identify the associated `Product` and the `Record` respectively.

The inclusion of `productId` in `RecordKey` is to allow records to be duplicated when a `Product` is Split. In the case of a Split, the `recordId` is the same on both copies of the record, but the `productId` differs for each output of the split. This allows the keys to remain unique.

6.1.1.2. Record Choices

Many choices are available on the Record contract. In the code excerpt is shown the choices `SetRequestObserverOnRecord` and `AcceptRecordHandover`. `SetRequestObserverOnRecord` allows the owner to set the `requestObserver` field. `AcceptRecordHandover` allows the `requestObserver` to set themselves as the product's current handler on the record. Example in-DAML usages of both choices can be found in *Product.daml,* and are discussed in the *DAML Implementation - Requests and Orders* section.

6.1.2. Product Contract

6.1.2.1. Product Fields

The `Product` contract references all records associated to a given product by means of the `recordKeys` field. The field is defined as a list of `RecordKeys`. Interestingly, the `recordKeys` field is not used when fetching a product's record history from the ledger. It exists on the `Product` with the aim of exposing convenient DAML choices that allow parties to perform operations on all associated records; for example, updating all a product's records when ownership is transferred. Fetching product records from the ledger is instead done by querying the ledger for all current records with a given `productId`.

`Products` are uniquely identified by means of a `productId` GUID. Additionally, individual versions of a `Product` contract can be referenced by a `version` field. The inclusion of the `version` field is not usually necessary, but aids UI rendering when the state of a component is dependant on a given `Product` version. The `ProductKey` is defined as `(Party, Text)`, with intended use for storing the `owner` and `productId` fields. The `version` field is not included in `ProductKey` because keys should only reference the current, unarchived version of the contract; old versions are archived when new `Products` are created. The implementation of all of the choices that update a `Product` must increment the `version` field.

As demonstrated in *Product State Transitions*, the `Product` is a stateful contract whose state is represented in the `productState` member. The type of `productState` is an enum, `ProductState`, that is enum defined at line 27 in *Product.daml* (see *DAML Excerpts* in the Appendices). When requests and orders progress by means of `Product` choice execution, `productState` is updated accordingly.

A product's request and order contracts are referenced from the `orderCid` field, defined as being of type `Optional OrderContractId`. `OrderContractId` is defined similarly to `RecordDetails` in *Records.daml.* It is a union-typed object that can hold a *tag/Cid* (Contract Identifier) pair. The `Optional` flag on `orderCid` indicates that the field is 'nullable'; able to store a value of `None` if no request or order currently exists for the product. When `orderCid` has a value, it is a contract identifier for the current request or order. This allows the request/order contract to be easily fetched.

6.1.2.2. Immediate Product Choices

`Product` choices that are not part of a request process are *immediate* choices. These include `AddProductionRecord`, `MergeProduct`, and `SplitProduct`. `AddProductionRecord` exists without a request because it is assumed that producers are product owners at the time of harvest. This, and the fact that there is no need for certification in this application, means that they do not require permission from anyone else to upload production records. Similarly, product merges and splits are done at the owners discretion and do not need require permission from other users.

The most interesting DAML code in the application is in the `MergeProduct` choice on the `Product` contract (*Product.daml* - line 75; see *DAML Excerpts* appendix). When merging two products together, it is required that both products are of the same product type, and are each currently owned and handled by the user. The output of a product merge is a product whose amount is equal to the sum of the previous two, of the same type, with merged labels and merged records.

The relative complexity of the implementation is in ensuring that merged labels and records do not contain duplicates. There is a (small) possibility that a product that is being merged with another shares some of the same records and labels. This would arise in the case where the two products were *split* from the same product, causing labels and/or records to be duplicated.

A 'real-world' example of this edge case is as follows. A producer creates a large batch of raw produce, splits it, and sends it to two different processors. The processors process the products independently. The two processed products are then sold to the same buyer, who then merges the products together. The two processed products contain the same production records, albeit different processing records.

The deduplication of labels is present at line 97. This is a fairly straightforward process, making use of DAML's `dedup` function. Slightly more complex is the merging of `recordKeys` from line 121 onwards. It is not enough to merge the lists and remove duplicates because there *are* no duplicated `RecordKeys`. When a product is split, the associated `Record` contracts are copied but with different keys, arising from the differing `productIds`. On merge, the merged records all ultimately need to reference the same `productId`. This means that the `recordKeys` of the `otherProduct` need to be updated with the `productId` of the `Product` from which the `MergeProduct` choice is taking place. This requires that the records associated to `otherProduct` need to update their `productId` fields via a loop (*Product.daml* - line 125) on the `OnProductMerge` choice on the `Record` (*Record.daml* - line 212). The loop runs on all the keys of the `otherProduct` that are not contained in `recordKeys` when `productId` is ignored. The process of 'ignoring' the `productIds` is present at lines 121 and 122; the keys are each mapped to exclude the second entry in the `RecordKey` triple. Finally, a new version of the product is created with updated keys, labels, and amount. The updated keys also contain a reference to a new *Merge Record* that is stored on the ledger to trace the merge.

6.1.2.3. Request-based Product Choices

Choices that initiate a process of request and approval are *request-based*. On the `Product` contract, requests can be initiated from the `MakeHandoverRequest`, `MakeSaleRequest`, `MakeTransportationRequest`, and `MakeProcessingRequest` choices.

In `MakeProcessingRequest` (*Product.daml* - line 239; see *DAML Excerpts* appendix), the `Product` is checked to ensure it is in the `IDLE` state. If it is in any other state, it is assumed that a request is already pending on the product, causing the choice execution to be aborted.

Request-based Product choices create request contracts. `MakeProcessingRequest` creates a `ProcessingRequest` contract. The `ProcessingRequest` is referenced from the product by assigning its contract identifier to the `orderCid` field when the `Product` is updated (*Product.daml* - line 263).

The choice gives the recipient of the request visibility of the `Product` and its `Records`. This is done by setting the `requestObserver` as the `serviceProvider` on the new version of the `Product` (*Product.daml* - line 260). After this, the referenced `Records` are updated via a loop (*Product.daml* - line 265) on the `SetRequestObserverOnRecord` choice (*Record.daml* - line 181).

6.1.3. Request and Order Contracts

Requests are created through choices on the `Product` contract, but are defined separately from the `Product`.

6.1.3.1. Request and Order Contract Fields

Each request contract holds fields important to the requested transaction. The `ProcessingRequest` is defined at line 967 of *Product.daml* (see *DAML Excerpts* in the appendix). Requests and Orders are not referenced by keys, but contract identifiers. This is because the execution of any choice on the request or order should intentionally make the contract unfetchable. A keyed contract is always fetchable provided that the contract is not archived without creating a new version. Contract identifiers point to individual, possibly archived, contract instances.

There is no 'parent' definition, `Request/Order`, like there is a `Record`. This is because the contents of the choice execution for each type of request/order differs depending on the type of request/order. This is not the case in `Record`, where the only available choices behave the same regardless of the stored record type.

6.1.3.2. Request and Order Contract Choices

All requests are implemented with three possible choices: WithdrawRequest, DeclineRequest, and AcceptRequest. On the `ProcessingRequest`, these are `WithdrawProcessingRequest`, `DeclineProcessingRequest`, and `AcceptProcessingRequest` (*Product.daml* - line 1016; see *DAML Excerpts* in the appendix).

Withdraw is a choice available to the party that made the request; the product owner. Decline and Accept are available to the service provider. In `ProcessingRequest`, `AcceptProcessingRequest` takes UI generated `recordId` and `processId` GUIDs, and a `recordTime` field. The `recordId` is used to create the Handover Record that results from the acceptance of the request. The `processId` is used to tie the Processing Start Record to the Processing End Record.

On request acceptance, the `requestObserver` needs to list themselves as the current handler of the product, both on the `Product` contract and all the referenced `Records`. In `AcceptProcessingRequest`, a loop (*Product.daml* - line 1053) on the `AcceptRecordHandover` choice (Record.daml - line 220) transfers the `requestObserver` to the `productCurrentHandler` field on the record. The `requestObserver` is listed as the handler of the `Product` contract when the `Product` is updated (*Product.daml* - line 1065).

Similar things happen in the Start and Finish choices of the New Order and Active Order contracts respectively. For brevity, examples are not discussed here or included in the code appendix.

6.2. DAML Unit Testing

DAML unit testing facilities are provided with the DAML SDK. The main form of unit testing is via *scenarios*. Scenarios are test runs of contract creation and choice execution that validate if the executed commands are legal and produce desired results.

The codebase contains two files dedicated to unit testing: *TestHelpers.daml* and *TestScenarios.daml*. *TestScenarios.daml* is the main unit test file and contains scenarios that test the outcomes of all the outcomes of the `Product` choices. *TestHelpers.daml* defines some simple helper functions that make the scenarios less cumbersome to write; for example, it defines a function for creating an initialised `Product` contract.

The general format of each unit test is as follows. First, the parties involved in each transaction are defined. Second, a `Product` contract is created, initialised in a way that is suitable for each individual test case; this `Product` is assigned to the `before` variable. Third, the choice under test is exercised. The choice returns a contract identifier to the newly created product. The contract identifier is used to fetch the resulting product, which is assigned to the `after` variable. The `before` and `after` products are checked and compared through a series of assertions. If at any time an assertion fails, or a transaction aborts for any reason, the test is considered to have failed.

The code appendix contains two DAML unit test examples: `test_MakeProcessingRequest` (*TestScenarios.daml* - line 124) and `test_AcceptProcessingRequest` (*TestScenarios.daml - line 124*).

In `test_MakeProcessingRequest`, the assertions on `before` and `after` are checking that the `MakeProcessingRequest` choice has transitioned the `Product` to a state that is indicative of the fact that there is a `ProcessingRequest` associated with the `Product`. To this end, it checks that `after.state` is set to `PROCESSING_REQUESTED`, that the `requestObserver` is set to the `processor`, and that the `orderCid` is set to the newly created `ProcessingRequest`'s identifier. It also checks that the version of `after` is incremented from `before`, and that the `recordKeys` have remained unchanged. Finally, the `processor` attempts to `fetch` all the `Records` and the `Product`, to ensure that they have been made observers on the `Product` and `Records`. If the `processor` has not been made an observer, the fetches will fail, causing the test to fail.

In `test_AcceptProcessingRequest`, the assertions start by checking that the `handler` on the `Product` has been updated, and that the `requestObserver` field has been cleared. It then checks that the creation of the `Record` has influenced the state of the `Product`. This is done by checking that the `completionTime` on the `Record` matches the `Product`'s `lastUpdated` field, that the `recordKeys` now contain the newly created Handover Record, and that the `lastRecordType` is a `HANDOVER`. Finally, the `processor` attempts to `fetch` all the records, and checks that they are no longer the `requestObserver` on the records, and that they are set as the `productCurrentHandler`.

6.3. User Interface Implementation

The UI is implemented in React and TypeScript, and uses Material UI components. Communication with the DAML ledger is performed using DAML's `@daml/ledger` node module. The DAML compiler's code generation feature was used to generate TypeScript interface definitions from the DAML source code. The implementation was based on the `daml-ui-template` project [24]. This template contains examples of DAML ledger interaction from a React/TypeScript frontend. A Redux store is used to store user state, including their access token.

Only selected UI source examples are discussed in this section. This limitation is due to the fact that the UI codebase is very large. The examples are mostly taken from components that interact with the DAML ledger.

6.3.1 General Ledger Interactions

In order for the client to interact with the ledger, the client must provide a JSON Web Token (JWT). The code that implements the token creation comes from the `daml-ui-template` code and is hence not discussed here.

To perform ledger operations with the `@daml/ledger Ledger` class, the client passes the token to the `Ledger` constructor, along with the HTTP and Web Socket (WS) base URLs. Once constructed, several functions available on the `Ledger` can be used, including `fetch`, `stream`, `create`, and `exercise`.

6.3.2 Fetching Ledger Data

The user interface regularly fetches data from the ledger in order to present ledger-stored product history to users. There are two notable variants of data retrieval that are available in the client libraries: *fetching* and *streaming*. *Fetches* are a one-off requests that returns all contracts that satisfy a provided query. *Streams* open web sockets that sends continuous updates to the client; for instance, by sending new contracts to the client when they are made available to the user. The `GenericTable` (*GenericTable.tsx* - provided in *TypeScript Excerpts* appendix) uses both streams and fetches.

`GenericTable` is the component used in all the main pages to display contract details to the user. It is implemented as a generic component to prevent having to rewrite the same functionality for each DAML contract type. The `GenericTable` has a React `prop`, `useStream`, that allows the calling code to specify whether or not records should be retrieved using a stream or a fetch. The conditional execution on `useStream` is in `componentDidMount` at line 57. `componentDidMount` is a React lifecycle method that is called when the component is first mounted into React's virtual DOM [25]. If `useStream` is set to true, the table will attempt to fetch records using a continuous stream; the stream being set up in `initialiseStream`, line 86. If false, it will fetch data in the `fetchData` method at line 69.

In the Products page, `useStream` is set to `true`. This is because the actions available on the Products page alter the table's contents, meaning that it ought to be dynamically updated by a continuous data stream. All other table pages do not need to be dynamically updated, and hence set `useStream` to `false`.

The implementation of `fetchData` is fairly straightforward. First, the `Ledger` is initialised using the user's token that has been retrieved from the Redux store. Then, a fetch is attempted using the `Ledger`'s `query` method. `query` takes a DAML-generated `Template` and an optional `query` parameter. The `Template` is generated from the DAML source, and specifies the type of DAML contract that the function should fetch. The `query` parameter allows field restrictions to be made to the fetch; for example, by querying for `Record` contracts that have their `recordType` field set to `PRODUCTION`. If the fetch is successful, the response payload is passed to `onRecordsReceived` (line 79). `onRecordsReceived` updates the component's local state with the received records, and sends the records to the call site of `GenericTable` if the `onRecordsReceived` callback `prop` is defined.

`initialiseStream` is relatively complex. It starts with the `Ledger` being initialised in the same way as it was in `fetchData`. It is then passed as an argument to the `Stream` constructor. `Stream` is *not* a class provided by the `@daml/ledger` module, but a helper class that holds a DAML stream internally. The main business logic of the stream is defined in the `onChange` method at line 91. The complexity arises from the need to update the currently selected item of the table. If the currently selected `Product`, for example, is updated, the component's state needs to be updated to reflect the change in the selection's version. If the state is not updated to hold the new version, the selection will be cleared when the product is updated.

6.3.3. Updating Ledger Data

Ledger data can be updated in two ways: through `create` commands, and through `exercise` commands. A `create` creates a new contract ‘from-scratch’. An `exercise` references an existing contract and updates the ledger through a choice on the contract. All ledger updates in the application take place in the editors. All of the editors can be found in the *src/Components/Editors* folder of the UI).

The editors all share a similar implementation and usage workflow. They all contain input fields that allow users to submit data to the ledger. On clicking *Okay*, the UI verifies the data contained in the fields. If the data is okay, the ledger update action is called. If the data is not okay, the invalid fields are highlighted red.

There is only one use of `create` in the application, and that is in *ProductEdit.tsx* (provided in the *TypeScript Excerpts* appendix). `ProductEdit` is the editor that is used to create new `Product` contracts. It is the only example of `create` because all other types of contracts are intended to be created through choices on the `Product`. `onOkay` (line 40) is called when the user clicks *Okay* in the form. `validateNewProduct` returns `true` if the fields of the product editor are valid, and `false` if not. If the fields are valid, `putToLedger` is called. `putToLedger` (line 31) initialises a `Ledger`, as standard. It then calls `create`, with the type of contract, `Product`, specified as the first argument, and the new product’s fields in the second argument. `prepareProductForLedger` (defined in *src/Types/Product.ts*) formats the data from the editor input fields such that DAML ledger can comprehend them; for example, by converting dates to the ISO format. If the submission is successful, the editor closes.

An example of the use of `exercise` can be found in *ProcessingFinishEdit.tsx* (provided in the *TypeScript Excerpts* appendix). `ProcessingFinishEdit` is one of the few editors that fetches from the ledger as well as updating it. `ProcessingFinishEdit` is used when a processor completes a processing order. It allows them to specify the output product’s label, output amount, and finishing time of the process. This editor renders a header table containing information about the processing order. This requires that the `ActiveProcessingOrder` relating to the processing job is fetched from the ledger. This fetch takes place in `componentDidMount` (line 56).

`ProcessingFinishEdit` calls `exercise` in `putToLedger` (line 77). `exercise` takes the choice as the first argument. In the case of `ProcessingFinishEdit`, the choice is `ActiveProcessingOrder.CompleteProcessingOrder`. The second argument is the contract identifier of the `ActiveProcessingOrder`; contract identifiers are required when exercising choices on a contract. Finally, it receives the choice arguments. Here, `choiceArgs` includes a `recordId`. All GUIDs in the application are generated in the UI using `generateGuid`. The editor closes once the choice executes successfully.

6.3.4. Role-based Conditional Rendering

It is not possible to specify party-specific access controls in a DAML ledger's configuration. This means that the role-based privileges in this application are instead 'enforced' in the UI. Access controls are enforced in the UI by only rendering controls that are available to a user based on their role. UI-only enforcement makes it possible for users to query the ledger's API with actions that *shouldn't* be available to them. This could be overcome by introducing app middleware that only considered contracts created by the 'right' parties. Nonetheless, enforcing rigid access controls is outside the scope of the project, and was not actually specified by *The Company*. The UI access controls in this application are more of a presentational feature.

The best example of role-based conditional rendering in the application is in *ProductActions.tsx* (provided in the *TypeScript Excerpts* appendix). `ProductActions` is a component that is rendered in the Product Audit Header, and contains the buttons available to a user based on their role and the product's current state. The example in the appendix provides the code for the conditional rendering of *transportation* and *merge* action buttons.

In `ProductActions`, a row of action buttons is rendered when the product is in the `IDLE` state (line 329). Each action button renders a dropdown menu when clicked, and is disabled when no actions are available. The buttons are rendered in `renderActions` (line 258), and the available actions are calculated in `getAvailableMenuActions` (line 135).

`getAvailableMenuActions` considers three things: the user's *implied* role, the user's *assigned* role, and the state of the selected `Product`. `canTransport` (line 165) and `canSubmitTransportRequest` (line 166) are the two transportation-related actions that can possibly be exercised from a product. `canTransport` should have value `true` if product is in the `IDLE` state, the user is both the handler and the owner, the product has some existing records (`notNew`), and the user has an assigned role of `TRANSPORTER`. `canSubmitTransportRequest` simply requires that the user is the `owner`, the product is `IDLE`, and has some existing records. `canMerge` (line 169) is derived from a slightly more involved calculation. It checks to see if there exists some product that, with respect to the selected product, is: of the same type, with different `productId`, owned and handled by the user, and with positive quantity. These calculated booleans allow the available `EditorTypes` to be pushed to the `availableMenuActions` object.

The resulting `availableMenuActions` is used to render the contents of the menu. If there are no available actions for an action group, e.g. transportation, the entire action group's menu is disabled. If the menu renders, it is anchored under the action button that has been clicked. This is done by setting the `anchorEl` of the `Menu` to `this.state.clickedAction?.element` (line 239). The state variable `clickedAction` is set when a button is clicked, and contains a reference to the `HTMLElement` of the clicked button in the `element` field.

6.4. Sample Data

In order to demonstrate the application's capability of handling a wide range of product histories, some sample data has been generated. The sample data is generated in *SampleData.daml* in the DAML folder. This file makes use of *SampleDataHelpers.daml*.

The titles of the subsections correspond to the product descriptions of the products described. Product histories can be viewed by logging in as their owner and going to the *Products* page.

6.4.1. Oranges (Sample Data)

Oranges (Sample Data) is a product with production records that has not yet been traded.

Oranges (Sample Data) is owned by *Farmer Fred.*

6.4.2. Coffee (Sample Data)

Coffee (Sample Data) is a product with a processing record that has not yet been traded.

- The product is produced by *Farmer Fred* .
- It is then processed by *Processor Polly* on a request-basis.
- The product is then handed back to *Farmer Fred*.

Coffee (Sample Data) is owned by *Farmer Fred*.

6.4.3. Ginger (Sample Data)

Ginger (Sample Data) is a product with a processing record that has been traded and transported.

- The product is produced by *Farmer Fred* .
- It is then processed by *Processor Polly* on a request-basis.
- The product is then handed back to *Farmer Fred*.
- *Farmer Fred* sells the product to *Buyer Barry*.
- *Buyer Barry* requests *Transporter Tim* to transport the product to him.
- *Transporter Tim* hands it over to *Buyer Barry*.

Ginger (Sample Data) is owned by *Buyer Barry*.

6.4.4. Bananas (Sample Data)

Bananas (Sample Data) is an unprocessed product that has been split and sold to two different buyers.

- The product is produced by *Farmer Fred.*
- It is then split in half with new descriptions "*Bananas (Sample Data) (1)"* and "*Bananas (Sample Data) (2)".*
- *Bananas (Sample Data) (1)* is sold to *Buyer Barry.*
- *Bananas (Sample Data) (2)* is sold to *Buyer Bonzo.*

Bananas (Sample Data) (1) is owned by *Buyer Barry*.
Bananas (Sample Data) (2) is owned by *Buyer Bonzo*.

6.4.5. Sugar (Sample Data)

Sugar (Sample Data) is a product that is produced and processed separately, then bought and processed together.

- *Sugar Cane* is produced by *Farmer Fred* and *Farmer Felicity*.
- *Farmer Fred* has *Processor Polly* turn it into *Sugar*.
- *Farmer Felicity* has *Processor Percy* turn it into *Sugar*.
- *Buyer Barry* buys the *Sugar* from both farmers
- *Buyer Barry* merges the two together
- *Buyer Barry* has *Admin Alex* turn it into *Sugar Cubes*.

Sugar (Sample Data) is owned by *Buyer Barry*.

7.1. Benefits and Drawbacks of DAML

7.1.1. The Language

DAML syntax is very simple, and this makes it quick to learn. The language is intentionally abstract and high-level, so much so that it is almost human readable. This is thanks to the intelligent use of keywords that allow DAML contracts to be thought of as a parallel to traditional contracts.

The high-level nature of DAML allowed for rapid iteration of DAML contract implementations. This was especially convenient considering that requirements from *The Company* took a few weeks to become clear, causing intermediate DAML files to become quickly redundant.

The language is powerful for developing smart contract applications where privacy is of the utmost importance; entities are not visible to any parties that are not contract stakeholders. This feature of DAML requires developers desiring publicly visible contracts to employ a few workarounds. Such a contract would have to include an observing party whose access token is made publicly available. Fortunately, the application in this project did not require that *everyone* view products and their records. That being said, the *Provenance* application hints at a 'globally auditable' ledger [16]. This could make DAML an unsuitable choice for such an application, if workarounds are to be avoided.

Some features that developers take for granted in more traditional languages are not present in DAML. For example, conditional execution in DAML is limited to ternary statements. It is also not possible to define complex class relations such as inheritance, or to define generic contracts. The inability to create generic contracts can result in unnecessary code duplication.

7.1.2. Client Libraries

A useful feature of DAML is that the compiler can generate TypeScript files from DAML source. These TS files contain type definitions of contracts and types defined in the project's DAML source code. This is useful when developing in a strongly typed frontend language, such as TypeScript, in ensuring correlation between backend and frontend data types.

Several node modules are available for DAML/JS developers. In this project, the main library used was `@daml/ledger`. This library contains the `Ledger` class; a helpful wrapper for accessing the backend ledger via the API.

The main downside of the client libraries is that some of the TypeScript type signatures are quite ugly. This can make programming cumbersome in places where, for example, user class members have to be defined with correct DAML-generated type signatures. This can be overcome with some TypeScript wizardry.

The most sophisticated client library is the `@daml/react` library. As the name suggests, this library is intended for use with React. Unforunately, the majority of the functions available in this library can only be used with functional React components. Out of personal preference, the project's React source is written using class-based components.

7.1.3. Development Tools

The development tools available to DAML developers are limited. Visual Studio Code is the only IDE that supports DAML scenarios and syntax highlighting. This is not much of a problem; VSCode is becoming an industry standard and is quite pleasant to use.

Unfortunately, both the in-IDE and compiler feedback is sub-optimal. Many error messages and warnings are unhelpful. Some syntax errors cause the entire file to be underlined with red-squiggly lines. Hovering over the red lines will sometimes produce unhelpful messages such as ‘parse error’.

The worst DAML feedback messages result from aborted transactions. The stack-traces do not reference the line number of the user source at which the transaction aborted, but nonetheless contain many lines displaying stack-trace from methods of non-user code. The reason for abortion is also rarely presented in a way that is easily comprehensible or useful. For example, if you create a contract with a duplicated contract key, the error message will not tell you what the duplicated key is. Even worse is when a party cannot find a contract. The message reads *‘…couldn’t find contract ContractId(….)’* and doesn’t even tell you what type of DAML contract couldn’t be found! Painful debugging then ensues where you have to comment out code until you find the source of the problem.

7.1.4. Developer Community

The size of the online community is small. That being said, the DAML forum is excellent. It is regularly monitored by DA employees, all of whom are very helpful. The vast majority of my questions were answered within a couple of hours, and with great detail.

7.2. Does this Application need a Blockchain?

In the *Blockchain* section, the concept of *chainwashing* is defined as the practice of using blockchains in applications without good reason. It is important to ask: *Does this application need a blockchain?*

Such a question should usually be asked much earlier in the development process; during system design, prior to implementation. Considering that this is an academic project with the intention of exploring DAML as a development tool, the question of blockchain suitability did not intervene with development.

As with many blockchain applications, the features of this application could be implemented using a traditional database architecture. Products and records could be stored in the database, with the smart contract execution code being ported to server code. The database architecture would make for easier enforcement of privilege-based access controls. This project enforced access controls in the UI only.

The main reason, if not the only reason, for having this application on a blockchain is that the immutability of records is guaranteed. In centralised databases, records can be edited retrospectively. This means that auditors cannot be entirely sure that records have not been altered in deceptive ways. The system designers *could* implement a centralised solution that does not expose methods for altering past records, but this requires that you *trust* the system provider.

The answer to the question depends on the demands and selling point of the application. If the intervention of system administrators is required, and record immutability needn't be guaranteed, then a blockchain may not be the best option. With the intent of creating a solution for tracking product records that are guaranteed to be immutable, this application *does* in fact benefit from a blockchain.

7.3. Project Reflection

7.3.1. Change in Objectives

The original intention for this project was for it to be a ‘technical’ project with the aim of exploring model-based development of smart contract blockchain applications. It was decided that the best way of assessing the ‘model-based’ development tools, i.e. DAML, was to use them in a ‘full-stack’ application. It slowly became clear that the implementation time for this project was going to be significant, which it was. This transformed the project into more of a ‘software implementation’ project.

The majority of development time was spent on the user interface. This was the experience shared by the Innover Digital developers [19] when developing their prototype supply chain application using a combination of DAML and React/TypeScript. This was, in part, due to the fact that DAML programming is at quite a high-level. It was also a result of the fact that the design of the user interface was incrementally improved through experimentation, and could have been indefinitely improved with new features and styling.

The additional development time meant that some of the original objectives of the project, as set out in the *Preliminary Report* were not met. For example, the project did not include any ‘exploratory development’ of DAML helper tools. In retrospect, some time spent on developing tools for DAML could have been useful. This is because there are many aspects of the DAML development experience that are unsatisfactory. These are discussed in *Benefits and Drawbacks of DAML*.

7.3.2. Application To-Dos

Given more time, there are some things that could be done to improve the application. These improvements are roughly divided into three categories: additional features, resolving technical debt, and frontend styling.

7.3.2.1 Additional Features

Some possible new features are listed here. These would help the application be more suited to a 'real-world' scenario.

- **Marketplace** - Currently, the application has a very simplistic trading mechanism. This could be improved by allowing users to 'post' available products to a marketplace, and have buyers view available items that match user-provided search criteria. Buyers and sellers could then do some back-and-forth to settle on an agreeable price.
- **Multiple processing inputs** - Processing services currently accept only one type of input product. In the real world, processing can often take multiple inputs. For example, to create chocolate, a chocolate factory may take cocoa powder, sugar, and milk. The creation of processing records should allow for multiple types of input products.
- **Certification** - Product standards could be enforced by allowing certifying parties to join the system. These certifiers could issue certificates to parties as verification that their activities are inline with environmental and ethical standards.
- **User sign-up** - A very important feature in a real-world scenario is to allow new users to sign up. The system currently only has a login page.
- **Description updates** - Allow users to update product description at any point. Currently only available on product merge/split.

7.3.2.2. Resolving Technical Debt

The application has incurred a lot of technical debt, partly as a result of a rushed development towards the end of the project. There are several places where code could be refactored and reused. For example, the UI's *Editor* components, e.g. *ProductEdit.tsx*, could possibly be genericised to reduce code duplication. Also, the coding style throughout the application codebase is not entirely consistent.

7.3.2.3. Frontend Styling

Some of the styling of the application is wanting. The main areas of unsatisfactory styling are in the presentation of data tables. The main table at the top of each page has very basic styling, and looks unimpressive when viewed on a small screen that produces resizing. The record entries in the product timeline also look fairly unexciting. That being said, I am generally satisfied with design of the timeline as a whole.

8. Conclusion

The main aim of this project was to assess DAML as a development tool. This was achieved by creating a full-stack application running on a DAML ledger. The requirements provided by *The Company* allowed the application to be demonstrative of the kind of applications that blockchain startups are creating. The fact that the implementation period took longer than expected only added to the practical experience gained.

DAML has proved itself as being a convenient tool for developers that are new to blockchain and are not entirely familiar with the technology. There are certainly areas where DAML developer experience could be improved, but it is still relatively early days. Nonetheless, DAML is a great language; simple, yet powerful.

9. Appendices

9.1. How to Run the App

9.1.1. Environment Setup

These instructions are for macOS; the preferred development/test operating system.

For the UI to look its best, run on a reasonably large screen; a laptop screen will cause component resizing.

Visual Studio Code
VSCode is an IDE that has the option for DAML syntax highlighting.
Download from https://code.visualstudio.com/download

DAML SDK
The DAML SDK includes the compiler and provides syntax highlighting for VSCode.
Follow the installation instructions at https://docs.daml.com/getting-started/installation.html
Don't forget to add the *bin* folder to the *PATH* variable (this step is also outlined in the installation instructions).

Yarn
Yarn was the package manager used for the UI development. Installing Yarn may also require you to install *Homebrew* (on macOS).
Yarn: https://classic.yarnpkg.com/en/docs/install/#mac-stable
Homebrew: https://brew.sh/

9.1.2. Building and Running the Application

1) Open a terminal
2) Run **`make build`** from **`/am1146/code/trunk/app`**
3) Run **`daml start`** from **`/am1146/code/trunk/app`**
4) Open a second terminal
5) Wait for the DAML server to start - (*'Started server:…'* message in the first terminal)
6) Run **`yarn start`** from **`/am1146/code/trunk/app/ui`**
7) Go to **`localhost:3000`** in your browser
8) Log in as *Farmer Fred*
9) If no sample data loads in the *Products* table straight away, wait and refresh the page

9.2. Abbreviations

BBO	Blockchain Business Network Ontology
BC	Blockchain
BPMN	Business Process Model and Notation
Cid	DAML Contract Identifier
DA	Digital Asset (Company)
DAML	Digital Asset Modelling Language
dApps	Decentralised Applications
DLT	Distributed Ledger Technology
ER Models	Entity-Relationship Models
GUID	Globally Unique Identifier
HTTP	HyperText Transfer Protocol
JS	JavaScript
JWT	JSON Web Token
PoW	Proof-of-Work
SC	Smart Contract
TS	TypeScript
UI	User Interface
UML	Unified Modelling Language
US	User Story
WS	Web Socket

9.3. Common Phrases

"The Company" - The real-world company that shared their real-world project requirements for me to base my academic project on.

"This (academic) project" - This project, as opposed to the student development project that is running alongside this and in conjunction with *The Company* to arrive at an actual prototype.

9.4. DAML Excerpts

9.4.1. ServiceRequest.daml

am1146/other/report_example_code/ServiceRequest.daml

```
module ServiceRequest where

template ServiceRequest
  with
    serviceProvider : Party
    client : Party
    auditor : Party
    serviceDescription : Text
  where
    signatory client
    observer auditor
    controller serviceProvider can
      Accept : ContractId ServiceOrder
        with
          startTime : Time
        do
          create ServiceOrder with
            serviceProvider
            client
            auditor
            serviceDescription
            startTime
      Decline : ()
        do return ()

 template ServiceOrder
   with
     serviceProvider : Party
     client : Party
     auditor : Party
     serviceDescription : Text
     startTime : Time
   where
     signatory client, serviceProvider
     observer auditor
     controller serviceProvider can
       Complete : ()
         do return ()
```

9.4.2. Record.daml

am1146/code/trunk/app/daml/Record.daml

```
type RecordCid = ContractId Record
type RecordKey = (Party, Text, Text)
..
data RecordType
  = UNDEFINED
  | PRODUCTION
  | PROCESSING_START
..
data RecordDetails
  = ProductionRecordDetailsTag ProductionRecordDetails
  | ProcessingStartRecordDetailsTag ProcessingStartRecordDetails
..
data ProductionRecordDetails = ProductionRecordDetails
  with
    product : ProductInfo
    location : Location
    plotId : Text
    time : Time
      deriving (Eq, Show)

data ProcessingStartRecordDetails = ProcessingStartRecordDetails
  with
    processId : Text
    inputProduct : ProductInfo
    startTime : Time
    location : Location
      deriving (Eq, Show)
..
template Record
  with
    recordType : RecordType
    productId : Text
    recordId : Text
    completionTime : Time
    details : RecordDetails
    actor : Party
    productOwner : Party
    productCurrentHandler : Party
    requestObserver : Optional Party
    offerObserver : Optional Party
  where
    signatory actor
    key (actor, productId, recordId) : RecordKey
    maintainer key._1
..
    controller productOwner can
      SetRequestObserverOnRecord: ContractId Record
        with
          newRequestObserver : Party
        do
          assert (isNone requestObserver)
          create this with requestObserver = Some newRequestObserver
..
      OnProductMerge : RecordCid
        with
          newProductId : Text
        do
          create this with
           productId = newProductId

    controller requestObserver can
      AcceptRecordHandover : RecordCid
        do
          create this with
            productCurrentHandler = fromSome requestObserver
            requestObserver = None
```

9.4.3. Product.daml - *Product*, *MakeProcessingRequest*

am1146/code/trunk/app/daml/Product.daml

```
type ProductCid = ContractId Product
type ProductKey = (Party, Text)

data OrderContractId
  = ProcessingRequestCid (ContractId ProcessingRequest)
  | NewProcessingOrderCid (ContractId NewProcessingOrder)
..
data ProductState
  = IDLE
  | PROCESSING_REQUESTED
  | PROCESSING_ACCEPTED
..
template Product
  with
    version : Int
    state : ProductState
    productId : Text
    isProcessed : Bool
    description : Text
    lastUpdated : Time
    lastRecordType : RecordType
    owner : Party
    handler : Party
    productInfo : ProductInfo
    recordKeys : [RecordKey]
    requestObserver : Optional Party
    offerObserver : Optional Party
    orderCid : Optional OrderContractId
  where
    signatory owner
    observer requestObserver, offerObserver
    key (owner, productId) : ProductKey
    maintainer key._1

    controller owner can
..
      MakeProcessingRequest: (ProductCid, ProcessingRequestCid, [RecordCid])
        with
          processor : Party
          location : Location
          outputType : Text
        do
          assert ((isNone requestObserver) && (isNone offerObserver))
          assert (state == IDLE)
          assert (productInfo.amount.quantity > 0.0)

          processingRequestCid <- create ProcessingRequest with
            productKey = (owner, productId)
            productId
            location
            inputProduct = productInfo
            processor
            outputType
            productOwner = owner
            currentHandler = handler

          productCid <- create this with
            requestObserver = Some processor
            state = PROCESSING_REQUESTED
            version = version + 1
            orderCid = Some (ProcessingRequestCid processingRequestCid)

          oldRecordCids <- forA recordKeys $ \k -> do
            exerciseByKey @Record k SetRequestObserverOnRecord with
              newRequestObserver = processor

          return (productCid, processingRequestCid, oldRecordCids)
```

9.4.4. Product.daml - *MergeProduct*

am1146/code/trunk/app/daml/Product.daml

```
controller owner can
..
  MergeProduct: (ProductCid, RecordCid, [RecordCid])
    with
      otherProductKey : ProductKey
      recordTime : Time
      recordId : Text
      newDescription : Text
    do
      (otherProductCid, otherProduct) <- fetchByKey @Product otherProductKey

      assert (state == IDLE)
      assert (owner == handler)
      assert (otherProduct.state == IDLE)
      assert (otherProduct.owner == owner)
      assert (otherProduct.handler == owner)
      assert (otherProduct.productInfo.typeId == productInfo.typeId)
      assert (recordTime > otherProduct.lastUpdated)
      assert (recordTime > lastUpdated)

      let recordKey = (owner, productId, recordId)
      let newAmount = Amount with
            unit = productInfo.amount.unit
            quantity = productInfo.amount.quantity + otherProduct.productInfo.amount.quantity
      let newLabels = dedup (productInfo.labels ++ otherProduct.productInfo.labels)

      recordCid <- create Record with
        recordType = MERGE
        productId
        recordId
        completionTime = recordTime
        details = MergeRecordDetailsTag MergeRecordDetails with
          oldProductA = productInfo
          oldProductB = otherProduct.productInfo
          newProduct = ProductInfo with
            typeId = productInfo.typeId
            labels = newLabels
            amount = newAmount
          time = recordTime
          oldDescriptionA = description
          oldDescriptionB = otherProduct.description
          newDescription
        actor = owner
        productOwner = owner
        productCurrentHandler = owner
        requestObserver = None
        offerObserver = None

      let otherKeysMinusProductId = map (\k -> (k._1, k._3)) otherProduct.recordKeys
      let recordKeysMinusProductId = map (\k -> (k._1, k._3)) recordKeys
      let otherKeysMinusProductIdUnique = otherKeysMinusProductId\\recordKeysMinusProductId

      otherProductRecordCids <- forA otherKeysMinusProductIdUnique $ \k -> do
        exerciseByKey @Record (k._1, otherProduct.productId, k._2) OnProductMerge with
          newProductId = productId

      let otherRecordKeys = map (\k -> (k._1, productId, k._2)) otherKeysMinusProductIdUnique

      productCid <- create this with
        productInfo = ProductInfo with
          amount = newAmount
          labels = newLabels
          typeId = productInfo.typeId
        recordKeys = recordKeys ++ otherRecordKeys ++ [recordKey]
        lastUpdated = recordTime
        lastRecordType = MERGE
        version = version + 1
        description = newDescription
        productId

      archive otherProductCid

      return (productCid, recordCid, otherProductRecordCids)
```

9.4.5. Product.daml - *ProcessingRequest*, *AcceptProcessingRequest*

am1146/code/trunk/app/daml/Product.daml

```
type ProcessingRequestCid = ContractId ProcessingRequest

template ProcessingRequest
  with
    inputProduct : ProductInfo
    outputType : Text
    productId : Text
    productKey : ProductKey
    productOwner : Party
    processor : Party
    location : Location
    currentHandler : Party
  where
    signatory productOwner
..
    controller processor can
..
       AcceptProcessingRequest : (ProductCid, NewProcessingOrderCid, RecordCid, [RecordCid])
         with
           recordId : Text
           processId : Text
           recordTime : Time
         do
           (productCid, product) <- fetchByKey @Product productKey

           assert (recordTime > product.lastUpdated)

           orderCid <- create NewProcessingOrder with
             inputProduct
             outputType
             productId
             productKey
             productOwner
             processor
             processId
             location
             handoverTime = recordTime

           recordCid <- create Record with
             recordType = HANDOVER
             productId
             recordId
             completionTime = recordTime
             actor = productOwner
             productOwner
             details = HandoverRecordDetailsTag HandoverRecordDetails with
               oldHandler = currentHandler
               newHandler = processor
               time = recordTime
               product = inputProduct
             productCurrentHandler = processor
             requestObserver = None
             offerObserver = None

           oldRecordCids <- forA product.recordKeys $ \k -> do
             exerciseByKey @Record k AcceptRecordHandover

           archive productCid

           let recordKey = (productOwner, productId, recordId)

           productCid <- create product with
             version = product.version + 1
             lastUpdated = recordTime
             recordKeys = product.recordKeys ++ [recordKey]
             lastRecordType = HANDOVER
             handler = processor
             state = PROCESSING_ACCEPTED
             requestObserver = None
             orderCid = Some (NewProcessingOrderCid orderCid)

           return (productCid, orderCid, recordCid, oldRecordCids)
```

9.4.6. TestScenarios.daml

am1146/code/trunk/app/daml/TestScenarios.daml

```
test_MakeProcessingRequest = scenario do
  producer <- getParty "producer"
  processor <- getParty "processor"
  productCid <- createProductWithAProductionRecord producer
  before <- submit producer do fetch productCid
  (productCid, requestCid, recordCids) <- makeProcessingRequest MakeRequestArgs with
    owner = producer
    productCid
    otherParty = processor
   after <- submit producer do fetch productCid

  assert (after.state == PROCESSING_REQUESTED)
  assert (fromSome (getProcessingRequestCid (fromSome after.orderCid)) == requestCid)
  assert (fromSome after.requestObserver == processor)
  assert (isNone after.offerObserver)
  assert (after.version == before.version + 1)
  assert (after.recordKeys == before.recordKeys)
  assert (after.productInfo.amount == before.productInfo.amount)

  processorFetch <- submit processor do fetch productCid
  records <- forA recordCids $ \recordCid -> submit processor do fetch recordCid
  request <- submit processor do fetch requestCid

  return ()
..
test_AcceptProcessingRequest = scenario do
  producer <- getParty "producer"
  processor <- getParty "processor"
  (productCid, requestCid) <- createProductWithAProcessingRequest producer processor
  before <- submit processor do fetch productCid
  let recordId = "recordId:test_AcceptProcessingRequest"
  let recordKey = (producer, before.productId, recordId)
  (productCid, orderCid, recordCid, recordCids) <- submit processor do
    exercise requestCid AcceptProcessingRequest with
      recordId
      processId = "processId:test_AcceptProcessingRequest"
      recordTime = addRelTime before.lastUpdated (hours 1)
  after <- submit processor do fetch productCid
  record <- submit processor do fetch recordCid

  assert ((after.handler == processor) && (before.handler == producer))
  assert ((isNone after.requestObserver) && (isSome before.requestObserver))
  assert (record.completionTime == after.lastUpdated)
  assert (after.lastRecordType == HANDOVER)
  assert (after.state == PROCESSING_ACCEPTED)
  assert (after.version == before.version + 1)
  assert (after.recordKeys == before.recordKeys ++ [recordKey])
  assert ((recordKey `elem` after.recordKeys) && (recordKey `notElem` before.recordKeys))
  assert (after.productInfo.amount == before.productInfo.amount)

  records <- forA recordCids $ \recordCid -> submit processor do fetch recordCid
  forA records $ \record -> do
    assert (isNone record.requestObserver)
    assert (record.productCurrentHandler == processor)
    return ()

  return ()
```

9.5. TypeScript Excerpts

9.5.1. GenericTable.tsx

am1146/code/trunk/app/ui/src/Components/DataDisplay/Tables/GenericTable.tsx

```
export class GenericTable<Record, Templ extends Template<any,any,any>>
      extends React.Component<GenericTableProps<Record, Templ>, GenericTableState<Record>> {
..
  private stream: Stream<Templ> | undefined = undefined;
  private mounted: boolean = false;

  componentDidMount() {
    this.mounted = true;
    if (this.props.useStream) this.initialiseStream();
    else this.fetchData();
  }

  componentWillUnmount() {
    this.mounted = false;
    this.stream?.close();
  }

  private fetchData() {
    const { user, template, query } = this.props;
    if (!user.loggedIn) return;
    const ledger = new Ledger({ token: user.token, httpBaseUrl, wsBaseUrl });
    ledger.query(template, query)
      .then(results => {
         const records = results.map(contract => contract.payload);
         this.onReceiveRecords(records);
      });
  }

  private onReceiveRecords(records: Record[], selected?: Selection) {
    selected = !!selected ? selected : this.state.selected;
    this.mounted && this.setState({ ...this.state, isLoading: false, records, selected });
    this.props.onReceiveRecords && this.props.onReceiveRecords(records);
  }

  private initialiseStream() {
    const { user, template, comparator, getId, getVersion, onRecordSelected } = this.props;
    if (!user.loggedIn) return;
    const ledger = new Ledger({ token: user.token, httpBaseUrl, wsBaseUrl });
    this.stream = new Stream(ledger, template, this.props.query);
    this.stream.onLive(() => this.mounted && this.setState({ ...this.state, isLoading: false}));
    this.stream.onChange(results => {
      const records = results.map(contract => contract.payload);
      let nextSelection: Selection = this.state.selected;
      if (this.state.selected.id) {
        const selectedRecord = records.find(record => getId(record) === this.state.selected.id);
        if (!selectedRecord) {
          onRecordSelected && onRecordSelected(undefined);
          nextSelection = nothingSelected;
        } else if (getVersion(selectedRecord) !== this.state.selected.version) {
          const id = getId(selectedRecord);
          const version = getVersion(selectedRecord);
          onRecordSelected && onRecordSelected(selectedRecord);
          nextSelection = { id, version };
        }
      }
      if (comparator) records.sort(comparator);
      this.onReceiveRecords(records, nextSelection);
    });
    this.stream.onClose(closeEvent => {
      console.error('streamQuery: web socket closed', closeEvent);
      this.mounted && this.setState({ ...this.state, isLoading: true });
    });
  }
..
}
```

9.5.2. ProductEdit.tsx

am1146/code/trunk/app/ui/src/Components/Editors/ProductEdit.tsx

```
export class ProductEdit extends React.Component<ProductEditProps, ProductEditState> {
  constructor(props: ProductEditProps) {
    super(props);
    this.state = { product: emptyProduct(props.user), validationError: false };
  }

  private putToLedger = () => {
    if (!this.props.user.loggedIn) return;
    this.setState({ validationError: false });
    const { product } = this.state;
    const ledger = new Ledger({ token: this.props.user.token, httpBaseUrl, wsBaseUrl });
    ledger.create(Product, prepareProductForLedger(product))
      .then(() => this.props.onClose());
  }

  private onOkay = () => {
    const { product } = this.state;
    if (validateNewProduct(product)) this.putToLedger();
    else this.setState({ validationError: true });
  }
..
}
```

9.5.3. ProcessingFinishEdit.tsx

am1146/code/trunk/app/ui/src/Components/Editors/ProcessingFinishEdit.tsx

```
export class ProcessingFinishEdit
       extends React.Component<FinishProcessingEditProps, FinishProcessingEditState> {
..
  private mounted: boolean = false;

  componentDidMount() {
    this.mounted = true;
    const { product } = this.props;
    if (!this.props.user.loggedIn) return;
    if (!product.orderCid
        || product.orderCid.tag !== ProductOrderCidTag.PROCESSING_ACTIVE) return;
    const ledger = new Ledger({ token: this.props.user.token, httpBaseUrl, wsBaseUrl });
    ledger.fetch(ActiveProcessingOrder, product.orderCid.value)
      .then(orderContract => {
        this.mounted && this.setState({
          ...this.state,
          order: orderContract?.payload,
          orderCid: orderContract?.contractId,
          label: orderContract ? generateLabelForTypeId(orderContract.payload.outputType) : ""
        });
      });
  }

  componentWillUnmount() {
    this.mounted = false;
  }

  private putToLedger = () => {
    if (!this.props.user.loggedIn) return;
    this.setState({ validationError: false });
    const { endTime, orderCid, quantity, label, order } = this.state;
    const ledger = new Ledger({ token: this.props.user.token, httpBaseUrl, wsBaseUrl });
    if (!orderCid || !order) return;
    const choiceArgs = {
      recordId: generateGuid(),
      recordTime: ledgerDate(endTime),
      recordOutputProduct: {
        labels: [label],
        amount: {
          quantity,
          unit: unitToString(processedProductTypeFromId(order.outputType).unit)
        },
        typeId: order?.outputType,
      }
    };
    ledger.exercise(ActiveProcessingOrder.CompleteProcessingOrder, orderCid, choiceArgs)
      .then(() => this.props.onClose());
  }

  private onOkay = () => {
    const { endTime, quantity } = this.state;
    const { lastUpdated } = this.props.product;
    if (endAfterStart({ endTime: endTime, startTime: lastUpdated })
        && truthyFields(this.state) && validateNumericString(quantity)) {
      this.putToLedger();
    }
    else this.setState({ validationError: true });
  }
..
}
```

9.5.4. ProductActions.tsx

am1146/code/trunk/app/ui/src/Components/ProductAudit/ProductActions.tsx

```
class ProductActionsComponent
      extends React.Component<ProductActionsProps, ProductActionsState> {
..
  private onClickAction = (event: React.MouseEvent<HTMLElement>, actionGroup: ActionGroup) => {
    this.setState({ clickedAction: { element: event.currentTarget, actionGroup } });
  }

  private onCloseMenu = () => {
    this.setState({ clickedAction: undefined });
  }

  private onClickMenuItem = (editorType: EditorType) => {
    this.props.setEditorType(editorType);
    this.onCloseMenu();
  }

  private getAvailableMenuActions(): AvailableMenuActions {
    const { user, product, allProducts } = this.props;
    let availableMenuActions: AvailableMenuActions = {
..
      transportation: [],
      merge: [],
..
    };

    if (!user.loggedIn) return availableMenuActions;

    const idle = product.state === ProductState.IDLE;
    const isOwner = user.loggedIn && user.username === product.owner;
    const isHandler = user.loggedIn && user.username === product.handler;
    const notNew = (product.lastRecordType !== RecordType.UNDEFINED);
..
    const isTransporter = hasRole(user, [UserRole.ADMIN, UserRole.TRANSPORTER]);
    const positiveQuantity = (+product.productInfo.amount.quantity) > 0.0;
..
    const canTransport = idle && user.loggedIn && isHandler
                         && isOwner && isTransporter && notNew;
    const canSubmitTransportRequest = idle && isOwner && notNew;
..
    const canMerge = isOwner && isHandler && allProducts.some(p => {
      const _isOwner = (p.owner === user.username)
      const _isHandler = (p.handler === user.username);
      const _differentId = p.productId !== product.productId;
      const _sameType = p.productInfo.typeId === product.productInfo.typeId;
      const _idle = p.state === ProductState.IDLE;
      const _positiveQuantity = (+p.productInfo.amount.quantity) > 0.0;
      return _isOwner && _isHandler && _differentId && _sameType
            && _idle && idle && _positiveQuantity && positiveQuantity;
    });
..
    if (canTransport) availableMenuActions.transportation.push(EditorType.TRANSPORTATION);
    if (canSubmitTransportRequest)
      availableMenuActions.transportation.push(EditorType.TRANSPORTATION_REQUEST);
    if (canMerge) availableMenuActions.merge.push(EditorType.MERGE);
..
    return availableMenuActions;
  }

  private renderMenuItem(available: EditorType[], editorType: EditorType, prompt: string) {
    if (!available.some(e => e === editorType)) return null;
    else return (
      <MenuItem key={editorType} onClick={() =>
        this.onClickMenuItem(editorType)}>{prompt}</MenuItem>
    );
  }

```

(continued on next page)

```
  private renderMenuItems(availableMenuActions: AvailableMenuActions) {
    if (!this.state.clickedAction) return null;
    switch (this.state.clickedAction.actionGroup) {
..
      case ActionGroup.TRANSPORTATION:
        return [
          this.renderMenuItem(availableMenuActions.transportation,
                              EditorType.TRANSPORTATION, "Start Transportation"),
          this.renderMenuItem(availableMenuActions.transportation,
                              EditorType.TRANSPORTATION_REQUEST, "Request Transportation")
        ].filter(Boolean);

      case ActionGroup.MERGE:
        return this.renderMenuItem(availableMenuActions.merge,
                                   EditorType.MERGE, "Merge Products");

..
    }
  }

  private renderMenu(availableMenuActions: AvailableMenuActions) {
    return (
      <Menu
        id="simple-menu"
        anchorEl={this.state.clickedAction?.element}
        keepMounted
        open={Boolean(this.state.clickedAction)}
        onClose={this.onCloseMenu}
        getContentAnchorEl={null}
        anchorOrigin={{
          vertical: 'bottom',
          horizontal: 'center',
        }}
        transformOrigin={{
          vertical: 'top',
          horizontal: 'center',
        }}
      >
        {this.renderMenuItems(availableMenuActions)}
      </Menu>
    );
  }

  private renderActions() {
    const { classes } = this.props;
    const availableMenuActions = this.getAvailableMenuActions();
    return (
      <React.Fragment>
        <div className={classes.actions}>
          <ButtonGroup>
..
            <Button
              color="primary"
              variant="contained"
              onClick={e => this.onClickAction(e, ActionGroup.TRANSPORTATION)}
              disabled={availableMenuActions.transportation.length === 0}
            >
              <LocalShipping />
            </Button>
            <Button
              color="primary"
              variant="contained"
              onClick={e => this.onClickAction(e, ActionGroup.MERGE)}
              disabled={availableMenuActions.merge.length === 0}
            >
              <CallMerge />
            </Button>
..
          </ButtonGroup>
        </div>
        {this.renderMenu(availableMenuActions)}
      </React.Fragment>
    );
  }

  render() {
    if (this.props.product.state === ProductState.IDLE) return this.renderActions();
    else return this.renderRequestButtons();
  }
}
```

10. References

[1] Nakamoto (2008) - *Bitcoin: A Peer-to-Peer Electronic Cash System* - https://bitcoin.org/bitcoin.pdf

[2] Deloitte (2020) - *Deloitte's 2020 Global Blockchain Survey* - https://www2.deloitte.com/content/dam/Deloitte/se/Documents/risk/DI_2019-global-blockchain-survey.pdf

[3] blockchain-council.org (2019) - *Top 10 Promising Blockchain Use Cases* - https://www.blockchain-council.org/blockchain/top-10-promising-blockchain-use-cases/

[4] bitcoinist.com (2017) - *97% of Blockchain Startups are 'Chainwashing'* - https://bitcoinist.com/chainwashing-r3-swanson-blockchain-hype/

[5] Szabo (1997) - *Formalizing and Securing Relationships on Public Networks* - https://firstmonday.org/ojs/index.php/fm/article/view/548

[6] Szabo (1998) - *Secure Property Titles with Owner Authority* - https://nakamotoinstitute.org/secure-property-titles/

[7] Buterin (2013) - *Ethereum Whitepaper* - https://ethereum.org/en/whitepaper/

[8] hackernoon.com (2018) - *Everything You Need to Know About Smart Contracts: A Beginner's Guide* - https://hackernoon.com/everything-you-need-to-know-about-smart-contracts-a-beginners-guide-c13cc138378a

[9] daml.com (2020) - *Reading DAML Smart Contracts for Non-Programmers* - https://www.blockchain-council.org/blockchain/top-10-promising-blockchain-use-cases/

[10] youtube.com (2020) - *DAML Channel* - https://www.youtube.com/channel/UCogfiTgCgVubpLQ1i1VmPPg

[11] asx.com (2020) - *CHESS Replacement* - https://www.asx.com.au/services/chess-replacement.htm

[12] accenture.com (2018) - *Accenture Deploys New Software License Management Application on Digital Asset's Distributed Ledger Platform* - https://newsroom.accenture.com/news/accenture-deploys-new-software-license-management-application-on-digital-assets-distributed-ledger-platform.htm

[13] docs.daml.com (2020) - *Templates* - https://docs.daml.com/daml/reference/templates.html

[14] blog.digitalasset.com (2016) - *Introducing the Digital Asset Modelling Language: A Powerful Alternative to Smart Contracts For Financial Institutions* - https://blog.digitalasset.com/blog/introducing-the-digital-asset-modeling-language-a-powerful-alternative-to-smart-contracts-for-financial-institutions#:~:text=WHY%20ISN'T%20DAML%20TURING,to%20write%20any%20possible%20program.

[15] deloitte.com (2020) - *Using blockchain to drive supply chain transparency* - https://www2.deloitte.com/us/en/pages/operations/articles/blockchain-supply-chain-innovation.html#:~:text=Blockchain%20can%20enable%20more%20transparent,or%20use%20by%20end%20user.

[16] provenance.org (2016) - *From shore to plate: Tracking tuna on the blockchain* - https://www.provenance.org/tracking-tuna-on-the-blockchain

[17] theguardian.com (2014) - *Revealed: Asian slave labour producing prawns for supermarkets in US, UK* - https://www.theguardian.com/global-development/2014/jun/10/supermarket-prawns-thailand-produced-slave-labour

[18] innoverdigital.com (2020) - *About Innover* - https://innoverdigital.com/about/about-innover/

[19] daml.com (2020) - *How we built a distributed application for supply chain visibility using DAML in 4 weeks* - https://daml.com/daml-driven/how-we-built-a-distributed-application-for-supply-chain-visibility-using-daml-in-4-weeks/

[20] daml.com (2020) - *DAML* - https://daml.com/

[21] Rocha, and Ducasse (2018) - *Preliminary Steps Towards Modelling Blockchain Oriented Software -* https://hal.inria.fr/hal-01831046/document

[22] Marchesi, Marchesi, and Tonelli (2018) - *An Agile Software Engineering Method to Design Blockchain Applications -* https://arxiv.org/pdf/1809.09596.pdf

[23] Seebacher and Maleshkova (2018) - *A Model-driven Approach for the Description of Blockchain Business Networks -* https://core.ac.uk/download/pdf/143481277.pdf

[24] github.com (2020) - *digital-asset/daml-ui-template* - https://github.com/digital-asset/daml-ui-template/projects

[25] react.org (2020) - *React.Component* - https://reactjs.org/docs/react-component.html