University of Manchester
School of Computer Science
Project Report 2022

Metal Supply
Blockchain-Based Supply Chain
Metal Recycling Application

Author: Kaloyan Terziev

Supervisor: Dr Richard Banach

Abstract

Metal Supply
Blockchain-Based Supply Chain
Metal Recycling Application

Author: Kaloyan Terziev

Blockchain is a relatively new technology that promises to disrupt and refine many business processes looking into the future. On the other hand, the supply chain is now essential to every business' success and growth. Over the past 2 years we have seen the long-term devastating effects of a small (on first glance) disruptions in supply and demand. This clearly indicates that new types of supply chains must be explored. Last but not least, recycling (in particular metal recycling) is an important part of creating a circular economy which will help preserve natural resources and reduce emissions because of the reduced need of metal mining, excavating, and transporting. Recycling can also reinforce supply chains, minimizing costs and protecting the environment.
The aim of this project is to combine leverage the latest developments in those fields and build a blockchain-based metal supply chain system for storing, querying and transferring agents and records credentials.

Supervisor: Dr Richard Banach

Acknowledgements

I would like to give special thanks to my supervisor for providing me with encouragement and support during the development of the project.

In addition, I would like to thank my parents for their unstoppable encouragement throughout the years at the university. I also want to acknowledge my partner for the emotional support and advice on the project's idea.

Contents

1 Introduction **6**
1.1 Motivation Behind the Choice . . . 6
1.2 Project Aim and Objectives . . . 6
1.3 Success criteria . . . 7
1.4 Report Structure . . . 7

2 Background **9**
2.1 Challenges . . . 9
2.2 How can Blockchain solve the challenges? . . . 9
2.3 Blockchain Architecture . . . 10
2.3.1 Hashes . . . 10
2.3.2 Transaction . . . 10
2.3.3 Asymmetric-Key Cryptography . . . 10
2.3.4 Ledgers . . . 11
2.3.5 Blocks and Chaining . . . 11
2.3.6 Blockchain Categorization . . . 12
2.3.7 Blockchain consensus . . . 13
2.4 Hyperledger Sawtooth . . . 13
2.4.1 Features . . . 14
2.4.2 Sawtooth Architecture . . . 14
2.4.3 Sawtooth Network . . . 14
2.4.4 Sawtooth Validator . . . 16
2.4.5 Sawtooth Consensus . . . 16
2.4.6 Sawtooth Application Components . . . 16
2.4.7 Transactions, Batches, and Blocks . . . 17
2.4.8 State . . . 18
2.4.9 Sawtooth Events . . . 18
2.5 Related Activity . . . 18
2.6 Chapter Summary . . . 18

3 System Design **19**
3.1 Design Requirements And Planning . . . 19
3.1.1 Permissioned vs Permissionless . . . 19
3.1.2 Interaction with the Blockchain network . . . 19
3.1.3 Off-Chain Storage . . . 19
3.1.4 Modular Design . . . 20
3.1.5 Client vs Server Signing Model . . . 20

3.2 Metal Supply Architecture Overview . . . 21
3.3 Application Design . . . 22
3.3.1 GDPR Compliance . . . 22
3.3.2 State Data Models . . . 22
3.3.3 Database Schema . . . 25
3.4 Choice of Consensus . . . 25
3.5 Chapter Summary . . . 26

4 Implementation **27**
4.1 Disclaimer . . . 27
4.2 Architectural Components . . . 27
4.2.1 Transaction Processor . . . 27
4.2.2 REST API . . . 28
4.2.3 Event Subscriber . . . 30
4.2.4 User Interface . . . 32
4.3 Application Endpoints and Workflows . . . 34
4.3.1 Endpoints . . . 34
4.3.2 Workflows . . . 35
4.4 Docker Containerization of Services . . . 36
4.5 Directed Graphs of Locations . . . 38
4.6 Chapter Summary . . . 39

5 Evaluation and Testing **40**
5.1 Performance Evaluation . . . 40
5.1.1 Reporting Database and Blockchain Performance . . . 40
5.1.2 Load Testing . . . 41
5.2 Testing . . . 43
5.3 Chapter Summary . . . 44

6 Conclusion **45**
6.1 Challenges and Future Work . . . 45
6.1.1 Data Model and Competitiveness . . . 45
6.1.2 Corporate Espionage . . . 46
6.1.3 Analysis and its Limitation . . . 46
6.2 Achievements . . . 46
6.2.1 Metal Product History Analysis . . . 46
6.2.2 Multifunctionality of the Project . . . 47
6.2.3 No Single Point of Failure . . . 47
6.3 Reflection . . . 47
6.4 Conclusion . . . 47

Bibliography **49**

A **53**

List of Figures

2.1 Generic Chain of Blocks [1] 12
2.2 Sawtooth Architecture [2] 15
2.3 Sawtooth Network[2] 15
2.4 Sawtooth Application Development[3] 17

3.1 Agent Data Model 23
3.2 Record Data Model 24
3.3 Reporting Database Schema 26

4.1 Application Architecture 28
4.2 Transfer Record Code Snippet 29
4.3 Vue.js Authentication with Vuex[4] 33
4.4 Web Application Structure 34
4.5 Retrieve Workflow 36
4.6 Create Workflow 37
4.7 Update Workflow 37
4.8 Component Dependencies 38

A.1 Home Page 54
A.2 Agents Page 54
A.3 Records Page 55
A.4 Profile Page 55
A.5 Record Page 56
A.6 Record Locations 56
A.7 Record History Analysis 57
A.8 Dependant Records 57

List of Tables

5.1 Record Retrieval 40
5.2 Agent Retrieval 41
5.3 Retrieve Records and Agents Load Test 41
5.4 Retrieve Records and Agents after Authentication Load Test 42
5.5 Log In Load Test 42
5.6 Register Agent Load Test 42
5.7 Create Record Load Test 43
5.8 Update Record Load Test 43

Chapter 1

Introduction

1.1 Motivation Behind the Choice

Rapid population growth and technological advancement have increased the purchase, use, and discarding of electrical and electronic equipment (EEEs), resulting in the generation of a massive amount of Waste from Electrical and Electronic Equipment (WEEE)[5]. Unlike other materials, metals are not biodegradable and have virtually an unlimited lifespan and the potential for unlimited recyclability [6]. Hence they are well suited for sustainable development goals [7]. If appropriately managed, recycling metal can provide numerous benefits for the environment in terms of energy savings, reduced volumes of waste, and reduced greenhouse gas emissions associated with energy savings [6]. For example, the energy saved using various recycled metals compared to virgin ore is up to 95% for aluminium, 85% for copper, 60 % for steel, 75% for zinc, and 90% for nickel [8]. Metal recycling also conserves natural resources by reducing the amount of ore needed to be mined, transported and processed.

All the advantages of recycling can be maximised by using a supply-chain system. The supply-chain systems provide better monitoring over materials for easier tracking throughout their movement and storage. Using closed-loop systems with recycling at the end of life provides a pathway to lower environmental impacts [9]. It also leads to a source of high-value critical metals, such as lithium, cobalt, and graphite, used in the manufacturing of lithium-ion batteries [9].

1.2 Project Aim and Objectives

The application Metal Supply aims to leverage the latest developments in the blockchain field and, in particular, those by the Hyperledger Foundation[1]. The project explores the advantages and disadvantages of a Permissioned Blockchain network of records and agents defining the metal products and their owners, respectively. The objectives of the project are as follows:

- Research challenges in supply chain and recycling, as well as the difficulties of circular economy sustainability.
- Define the intended result of the project.

[1]Hyperledger Foundation hosts a number of enterprise-grade blockchain software projects. The projects are conceived and built by the developer community for vendors, end user organisations, service providers, start-ups, academics and others to use to build and deploy blockchain networks or commercial solutions.

- Research permissioned and permissionless blockchain structures and relevant consensus algorithms.
- Choose an appropriate blockchain framework based on the project scope, objectives and blockchain type.
- Familiarize myself with the chosen blockchain framework.
- Design the architecture of the blockchain system.
- Implement the back-end services within Docker[2] containers for easier distributing and deployment of the components.
- Design and implement a user interface to provide effortless access to the blockchain network.

1.3 Success criteria

The following metrics will measure the project's success:

- Leverages all beneficial aspects of a blockchain network but abstracts away its use for end-users.
- Register and login using email and password but create blocks and sign transactions using public/private keys.
- Enter, store and display data about metal products and owners.
- Display statistics and locations about the metal products, as well as statistics and locations of all metal products that led to the production of a given metal product throughout the whole supply chain. For example, a single aluminium can comes from 50% recycled cans, 23% ore, 11% recycled aeroplane parts, 9% recycled window frames, 4% recycled kitchen utensils, and 3% recycled foils.
- Create a platform which is versatile enough to be run as a private network for one company, a permissioned network for a group of trusted companies, or globally with open/restricted access

1.4 Report Structure

A systematic approach was used in the process of writing this document. The report takes us from the background knowledge needed for the project through the system design and implementation and finally to the evaluation and conclusion. Those parts have been discussed in detail in the corresponding chapters:

Chapter 2: Background provides the needed context behind all the technologies used along with an explanation of key terms. Those concepts will be essential in the evaluation of

[2]Docker is an open source containerisation combining application which enables developers to package applications into containers-standardized executable components

the project. The chapter explains how blockchain technology works and the architecture of Sawtooth applications. It also outlines the related activity in the decentralised supply chain for recycling metal products.

Chapter 3: System Design takes us through the design requirements and planning of the project execution. The chapter also gives us the application design steps as well as an overview of the finalised high-level architecture and the motivation behind it.

Chapter 4: Implementation gives us the details of building the four essential components of the system. The chapter also provides the containerisation strategy for the different services.

Chapter 5: Evaluation and Testing provides an analysis of different evaluations of the system, including performance and load testing. It also outlines the types of testing the system went through while in development.

Chapter 6: Conclusion takes us through the challenges and future improvements of the system, as well as a reflection on the project and, finally, a summary.

Chapter 2

Background

2.1 Challenges

There are many challenges in the supply chain of WEEE which prevents industries from increasing the effectiveness of their manufacturing[10, p. 190]. A few of the challenges in front of the circular WEEE management supply chain are the number of various stakeholders involved in the process, as well as the current traceability, transparency and quality of the reported data [10, p. 190]. Another challenge is the lack of control and ownership of WEEE. Producers are dependent on retailers and municipalities to hand over the WEEE to the producer or the organization acting on behalf of the producer [11]. Only at that point, producers are able to organise the control over the WEEE stream and can take the responsibility for the appropriate collection and recycling [11].

2.2 How can Blockchain solve the challenges?

Distributed Ledger Technologies (DLT) can introduce new ways to solve those challenges. Technology better known as Blockchain (a specific subset of DLT) enables tracking and monitoring of incremental transactions and provenance of assets throughout their lifecycle even in an adversarial environment, where participants do not trust each other ex-ante[12]. Blockchain can also be used to record parameters of an asset which are written permanently on the Distributed ledger. Those parameters, such as location or temperature, can be recorded using IoT devices for more accurate and consistent data. All records in the blockchain are immutable, which enhances the transparency and trust between the parties. Any changes in records, such as parameters, are written as consecutive blocks on the blockchain and can be validated by all participants in the network. Transfers of ownership are also effortlessly and securely done by blockchain systems, easily keeping track of all records' owners for reporting use cases. Blockchain systems can align supply chains with circular economy sustainability through traceability and transparency. It can make the information about the origin of a product, processes and the parties involved in related transactions and logistics visible, traceable, and verifiable by all those in the supply chain[13]. Blockchain systems can support the application of sustainability criteria for the selection of suppliers, vendors, materials, and products, as well as the design of more sustainable logistics networks and internal operations. [14]

Therefore, blockchain-based documentation of the product history increases transparency in the supply chain for a circular economy, providing information at the point of sale to build

trust and security regarding transactions between the involved parties [12].

2.3 Blockchain Architecture

Blockchain systems use a variety of fundamental principles taken from computer science (linked lists, distributed networking), cryptography (hashes, digital signatures, public-private keys) and finances (ledgers) seen from an abstract point of view[1]. Put together, they create the state-of-the-art technology we know today.

2.3.1 Hashes

Hashing refers to the process of generating a pseudo-unique fixed-size output from an input of nearly any size. The input could be, for example, a number, text, file, or even an image. The output is a sequence of characters that, although connected with the input, could not be reversed computationally efficiently. Those functions are called one-way hash functions strictly for that ability. Hash functions are also designed to be collision-free, meaning that it is theoretically possible for a duplicate hash value from 2 different inputs. It is highly unlikely and computationally infeasible.

A typically used algorithm for hashing in many blockchain applications is the Secure Hash Algorithm (SHA), with an output size of 256 bits (SHA-256) [1]. It generates 32 8-bit characters meaning that there are 2^{256} different outputs. The specification for the algorithm can be found in Federal Information Processing Standard [15].

2.3.2 Transaction

A transaction in the context of a blockchain system is the act of transferring a digital asset from one owner to another. It could be a digital currency, digital painting, or the project's purpose - a metal product. A typical transaction could include:

- **Account IDs** of the current and future owners. Most blockchain systems use their public keys, but other indicators could be used as well.
- **Asset ID** subject to being transferred. Only relevant for blockchain systems with multiple assets. Cryptocurrencies do not use fields of this type because their asset is the cryptocurrency itself. A product barcode might also be relevant for some use cases.
- **Amount**. The amount of the digital asset to be transferred.
- **Transaction ID**. A unique identifier for the transaction. Some blockchain systems use the hash value of the whole transaction for transaction ID.

A transaction should also be easily verifiable by blockchain nodes. That's why transactions are also signed and can be easily verified with public/private key cryptography.

2.3.3 Asymmetric-Key Cryptography

Asymmetric-key cryptography or public/private key cryptography utilises two keys - private and a public key which are mathematically related to each other[1]. Although there is a relation between the two keys, it is computationally infeasible to determine the private key from the

public key. That is why the public key could be shared publicly without reducing the cryptographic security. When data is digitally signed, the private key generates a signature. Provided with the public key and the signature, one can easily verify the signature.

In the context of blockchain networks, asymmetric-key cryptography allows a node to easily verify whether the user that transfers a digit asset to another user is in possession of the private key capable of signing the transaction[1].

2.3.4 Ledgers

A ledger is a collection of transactions. Nowadays, they are usually kept digitally and very often in large databases that are owned by a centralised trusted organisation. The organisation authenticates, authorises, maintains and modifies the transaction records on behalf of a community of users. Private ledgers do not come without their disadvantages. The user base must trust that transactions are correctly validated, the list of transactions is complete containing all transactions that have been received and that the transaction history hasn't been altered. The user base must also trust that the organisation is properly backing up their data in case of a data loss.

A ledger implemented using a blockchain can mitigate these issues by using a distributed consensus mechanism[1]. The blockchain ledger will be copied and distributed amongst every node within the system[1]. New transactions are submitted to a node which then redistributes the transaction around the network for validation. If the new transaction is validated, it is written on the blockchain. Due to the fact that the network is distributed, there are many copies of the ledger. Even if a data loss occurs at one node, the ledger is backed up by many other nodes. The validation is undertaken by many nodes which prevents the possibility of poorly validated transactions. Every block written on the blockchain can never be modified, preventing issues with altered or missing transactions.

2.3.5 Blocks and Chaining

When submitting transactions to some or all nodes in the network, it does not mean they will be written on the blockchain immediately. Initially, they are queued in a mining pool, then given to the mining node for validation and later added to the blockchain. A typical block consists of several data fields listed below.

- **The block number**. Starting from the genesis block, the consecutive number of the current block.
- **The current block hash value**. The hash output from giving all data fields from the block to a hash function, including the previous hash value.
- The previous block hash value
- **The Merkle tree root hash**. It is calculated by recursively hashing the pair of hash values of adjacent transactions until only one is left. That way, only one hash value is saved instead of all transaction hashes. It is also used for validating if the transactions have been altered, as only a bit change in one transaction will alter the final Merkle tree root hash.
- **A timestamp**. The time the block was written on the blockchain

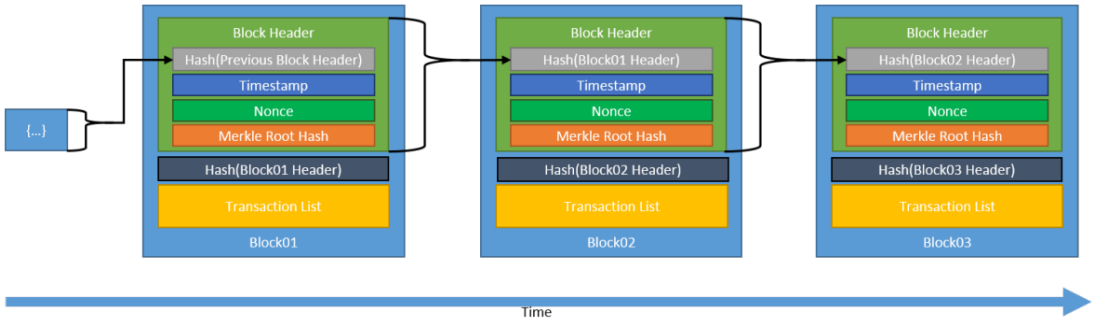

Figure 2.1: Generic Chain of Blocks [1]

- **The size of the block**. The sum of sizes of block data fields
- **The nonce value**. A number guessed by the mining node to solve the hash requirement set by the blockchain. Typically used in cryptocurrencies which use the PoW consensus mechanism.
- **A list of transactions**. The transactions included within the block.

Blocks are chained together through each block containing the hash of the previous block's header, thus forming the blockchain[1]. If a previously published block were changed, it would have a different hash[1]. This would also cause all subsequent blocks to have different hashes since they include the previous block's hash[1]. Making it possible to detect and reject any changes to previously published blocks easily[1]. Figure 2.1 shows a generic chain of blocks.

2.3.6 Blockchain Categorization

There are two main categories of Blockchain - Permissioned and Permissionless. The main difference between them is the visibility of the blockchain.

In permissionless blockchains, everyone could become a node and start validating transactions. Everyone has access to all of the information in the ledger; in other words - all transactions written on the blockchain are publicly visible. Cryptocurrencies use permissionless blockchains because they aim for global adoption among all people of a given digital currency. The primary consideration in this kind of blockchain is what happens if the node acts maliciously. Cryptocurrencies often provide incentives (digital currency) for users to act reasonably and follow the blockchain protocol rules. However, some users choose to act maliciously if that leads to a greater reward. This could happen by malicious users getting enough power (processing power, stake, etc.). That way, they could prevent transactions from being written or even submit an alternative chain which attacks the concept of "immutability" within a blockchain system [16].

In permissioned blockchains, only a group of selected users has read/write access to a given blockchain. It is primarily used between enterprises that desire to work together but do not necessarily trust each other. To enforce the trust, a permissioned blockchain can allow multiple companies to cooperate with each other without centralising the management of the data in one of those companies. Permissioned blockchains can be set up to allow only a group of users to write transactions, whereas another group can view the data on the blockchain. In

some use cases, it might be relevant to allow public visibility of the blockchain data while only permitting specific users to record transactions. Successful applications of the permissioned blockchains can be observed in banking, supply chain, insurance and healthcare.

2.3.7 Blockchain consensus

In a blockchain system, there are generally many mining nodes that have an incentive (digital currency, assets, etc.) to write a block on the blockchain. However, many mining nodes are involved in the validation of a block. Why would those nodes validate a block that will reward another node when they can reject the block and only verify the blocks validated by them to receive the reward? Blockchain systems have various methods for dealing with this malicious behaviour called consensus mechanisms. Listed below are the most used and well-adopted consensus models.

- **PoW** - In the Proof of Work consensus algorithm, mining nodes compete by solving a computationally intensive puzzle for having the right to publish a new block on the blockchain. The puzzle requires finding a block's hash value by adjusting a block field called the nonce. The mining nodes brute force an acceptable nonce value that will generate a hash value according to a given requirement (e.g. a hash value starting with four zeroes). Remember that even a 1-bit change in one of the fields results in a massive shift in the resulting hash. Using this principle, nodes prove to other nodes they have exchanged a specific computational power for the incentive for publishing a block. If a node decides to be malicious and reject a valid block, the chances are that it might not be able to reveal another nonce in time before other nodes validate the block and write it on the blockchain.

- **PoS** - The Proof of Stake consensus mechanism relies on the fact that the more stake, or in other words, staked cryptocurrency that cannot be used, the user has, the less likely it is that the user will harm the system. The Proof of Stake does not rely on vast processing power to allow block creation compared to the PoW. Consequently, it requires less computational resources and electricity to be run. Under proof of stake systems, the "rich" can more easily stake more of the digital assets, earning themselves more assets; however, obtaining the majority of assets within a system to "control"is generally cost-prohibitive[1].

- **Round Robin** - The round-robin consensus is adopted when there is a given amount of trust between the parties in the blockchain system. It is generally used in permissioned blockchains where nodes take turns writing the block on the blockchain.

2.4 Hyperledger Sawtooth

Hyperledger Sawtooth is an enterprise solution for building, deploying, and running distributed ledgers. As noted by the Sawtooth documentation[2], the Hyperledger project's architecture is highly modular and allows applications to choose the transaction rules, permissioning, and consensus algorithms that support their unique business needs[2].

2.4.1 Features

The Sawtooth documentation[2] emphasises on the following distinct features of Sawtooth:

- **Separation between the application level and the core system.** Sawtooth's modular design lets one focus on business rules without needing to know the underlying structure of the system. Transaction processors handle the application's server-side business logic while validators manage the core functions of verifying transactions and reaching consensus. [2]

- **Global state agreement.** Sawtooth offers cryptographically verifiable distributed ledgers with global state agreement, which is an assurance that each node has cryptographically identical copies of the blockchain database.[2]

- **Dynamic consensus algorithms.** The Sawtooth documentation defines consensus as the process of building agreement among a decentralised group of mutually distrusting participants. Sawtooth isolates consensus from transaction semantics. More importantly, the Sawtooth documentation indicates that different types of consensus are allowed on the same blockchain[3]. The consensus is selected during the initial network setup and can be changed on a running blockchain with a transaction.[3]

- **An event system that supports creating and broadcasting events.** As noted by the Sawtooth documentation, applications that use Hyperledger Sawtooth can:

  - Subscribe to blockchain-related events, such as committing a new block or switching to a new fork.
  - Subscribe to application-specific events defined by a transaction family.
  - Relay information about transaction execution to clients without storing that data in state.

- **On-chain governance for permissions and configuration settings** Private networks of Sawtooth nodes can be easily deployed with on-chain permissioning. The blockchain stores configuration and permission settings, such as roles and identities, so that all participants in the network can access this information.[2]

2.4.2 Sawtooth Architecture

Sawtooth separates core blockchain functions from business logic through transaction processors and client applications, as indicated in the Sawtooth documentation[2]. See Figure 2.2.

2.4.3 Sawtooth Network

A Sawtooth Network consists of a set of validator nodes. A node is a host system that runs the validator process and one or more transaction processors. All nodes in the Sawtooth network are peers, and thus there is no concept of a master or head node, as noted by the Sawtooth documentation[2]. The first validator node will be no different from the other except that it will create the Genesis block, distributed to all nodes. Figure 2.3

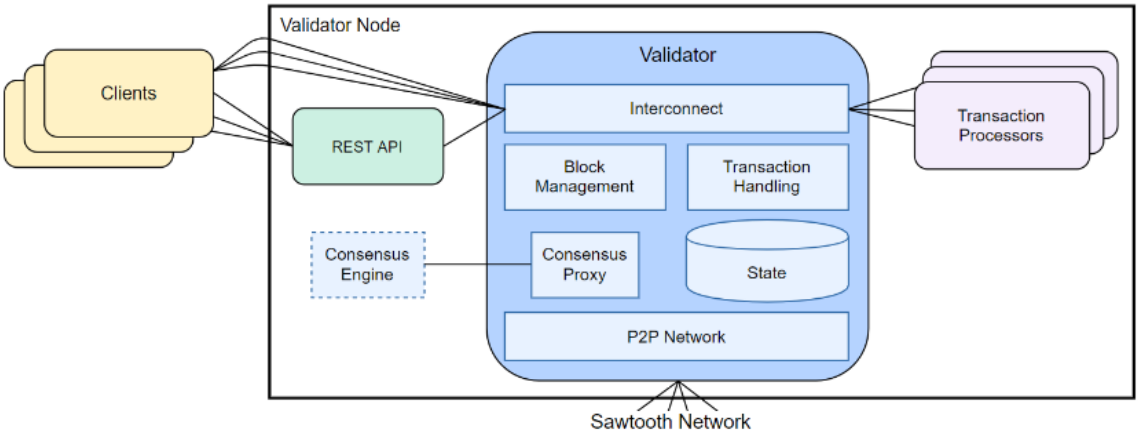

Figure 2.2: Sawtooth Architecture [2]

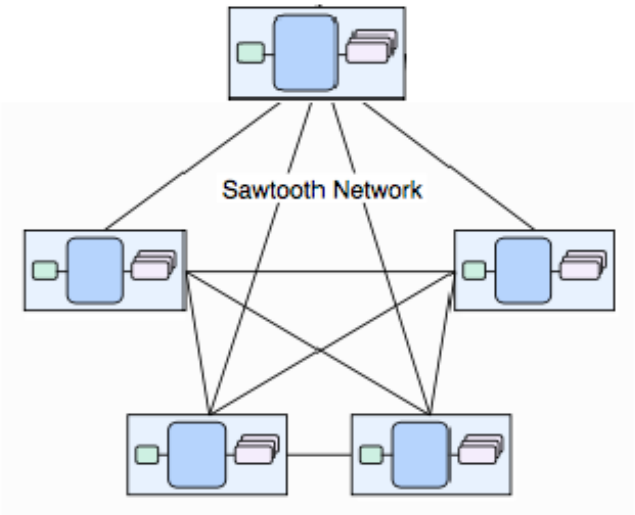

Figure 2.3: Sawtooth Network[2]

2.4.4 Sawtooth Validator

As indicated by the documentation, the Sawtooth validator is a core component that:

- validates batches of transactions,
- combines batches into blocks,
- maintains consensus with the Sawtooth network for adding (candidate) blocks to each node's version of the blockchain
- coordinates communication between clients, transaction processors and other validator nodes.

Each validator has its own instance of the blockchain and communicates with the other validators using a peer-to-peer network. The gossip protocol enables the communication between the validators

2.4.5 Sawtooth Consensus

Sawtooth supports dynamic consensus, meaning the consensus algorithm can be changed without reinitialising the blockchain or restarting Sawtooth, as noted by the Sawtooth documentation[2]. Sawtooth currently supports these consensus implementations:

- **Proof of Elapsed Time (PoET)**, a Nakamoto-style (lottery) consensus algorithm that is designed to be a production-grade protocol capable of supporting large network populations. PoET relies on secure instruction execution to achieve the scaling benefits of a Nakamoto-style (lottery) consensus algorithm without the power consumption drawbacks of the Proof of Work algorithm.
- **PoET simulator**, which provides PoET-style consensus on any type of hardware, including a virtualized cloud environment.
- **PBFT**, a leader-based, non-forking consensus algorithm with finality that provides Byzantine Fault Tolerance (BFT). Ideal for smaller, consortium-style networks that do not require open membership.
- **Dev mode**, a simplified random-leader algorithm that is useful for development and testing.

2.4.6 Sawtooth Application Components

A Sawtooth application typically includes the following components, as noted by the Sawtooth documentation[2]. See Figure 2.4:

- **A data model** to define the valid operations and specify the transaction payload (the data for the application that is given to the Validator node)
- **A Transaction Processor** to define the business logic of the application. It validates batches of transactions and updates the state based on the rules (part of the business logic) defined by the application. A transaction processor runs on every Validator node in the Sawtooth network.

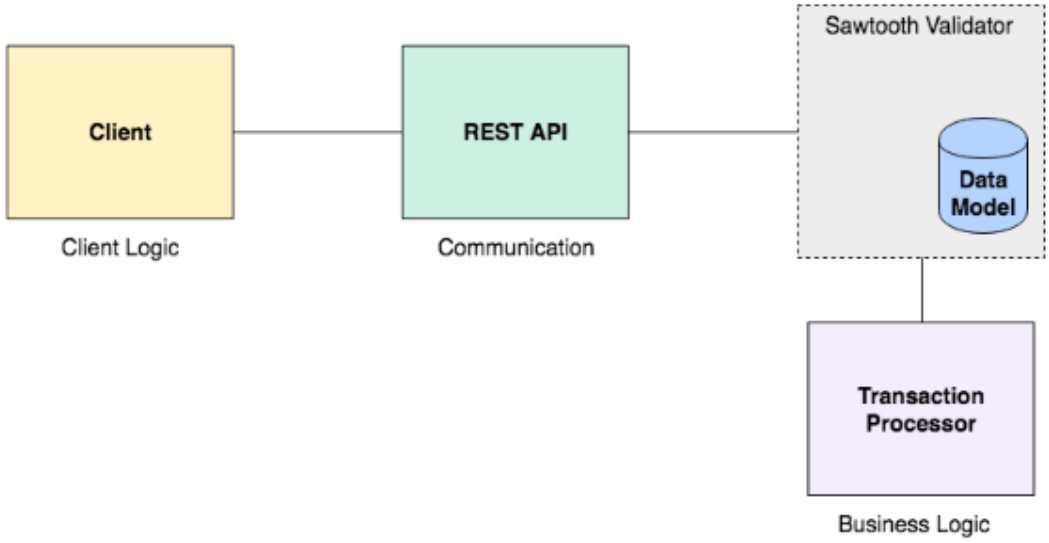

Figure 2.4: Sawtooth Application Development[3]

- **A client** to display data and generate and send transactions(changes to the state) to the validator nodes
- **A REST API** (optional) to handle communication between the client and Validator nodes. It can simplify client development by adapting client-validator communication (typically done via the ZeroMQ protocol) to standard HTTP/JSON.

2.4.7 Transactions, Batches, and Blocks

State changes occur when a Sawtooth Validator commits a valid block containing a batch of transactions to the blockchain, as noted by the Sawtooth documentation[2]. Firstly, a client creates one or more transactions, wraps them in a batch, and submits the batch to a Validator Node (usually through a REST API). According to the business logic, the Validator node verifies the batch (according to the business logic) and writes them in a block, and commits it to the blockchain. At that point, transactions in the block cause the blockchain state to change.

- **Transaction**: A single change in the state of the blockchain. Each Sawtooth transaction includes a transaction header that identifies the signer (the client that created the transaction) and a unique transaction ID.
- **Batch**: A transaction is always wrapped inside of a Sawtooth batch, which is the atomic unit of state change. All transactions within a batch are either committed to state together or are not committed at all. Each Sawtooth batch has a batch header that includes the public key of the client who created the batch (often the same as the signer) and the application name (transaction family name) and version
- **Block**: A group of Sawtooth batches. A block has a header that includes a timestamp, a signer, and a hash (unique block ID). After a block is committed, the header also identifies the previous block in the blockchain.

2.4.8 State

State is the Sawtooth blockchain, as defined by the Sawtooth documentation[2], represented as a log of all changes that have occurred since the genesis block. Each validator node has its own copy of the state in a local database. Each validator updates its state database to reflect the current blockchain status as blocks are published.

Global State represents the consensus of all participants on the state of the blockchain.

2.4.9 Sawtooth Events

A Sawtooth application can subscribe to core Sawtooth events and application-specific events, as defined by the Sawtooth documentation[2], that occur on the blockchain, such as a new block being committed or switching to a new fork, then informs clients about the transaction execution without storing that data in the state. An application can also request event catch-up by issuing a catch-up subscription request with a specific block ID. The Sawtooth validator sends data for all events that have occurred after that block was committed.

2.5 Related Activity

Although blockchain technology is in its early phases, its supply chain application has been widely researched. However, there are not many articles which focus their attention on sustainability as well. There are a few articles regarding blockchain as the future of sustainable supply chain management[17], or enabling a circular economy through blockchain technology[18] but none of them focus on recycling metals. There are also other articles describing the traceability of metals using blockchain but in the context of EV batteries[19]. However, none of those studies have focused their attention on blockchain backed supply chain systems for recycling metal products.

Outside the world of academic papers, there is a platform called RecycleGO aiming to bring transparency and better decision-making to the recycling supply chain using blockchain technology[20]. Their Blockchain system tracks entire life cycle of recycled materials. However, my project focuses exclusively on metal products, whereas RecycleGO focuses their attention primarily on plastic bottles. Although very similar to my project, I wasn't able to find any details about their platform because it is a closed-source software.

2.6 Chapter Summary

This chapter provided the background knowledge needed to fully comprehend this document. It discusses the challenges in supply chain and recycling and how blockchain can address them. It also ensures readers are confident with the use of various terms and concepts related to blockchain systems and the Sawtooth framework.

Chapter 3

System Design

This section covers the design decisions taken in the planning phase of the project.

3.1 Design Requirements And Planning

3.1.1 Permissioned vs Permissionless

The Metal Supply application is meant to be a platform that meets companies' supply chain needs. In addition, it is focused on recycling products and creating closed-loop systems for recycling products. With regards to having different companies cooperating with each other, the choice of blockchain category was straightforward. It needs to be a permissioned network for the possibility of having different levels of visibility of the blockchain data that does not need to be necessarily public. Having permission to record transactions on the blockchain should be kept only in the hands of the people managing their companies' supply chain. This permission could also be used by IoT devices that can record relevant information for the supply chain, such as location.

3.1.2 Interaction with the Blockchain network

The application will certainly need a client-facing user interface that will be used for supply chain management. Most front-end technologies will suffice regarding a simple, easy-to-use and intuitive user interface. However, integration with Google Maps is required to display the location data for maximum engagement from the supply chain managers' side. Hence, a decision was made in the favour of the popular JavaScript[1] framework - Vue.js[2]

3.1.3 Off-Chain Storage

The application also needs a reporting database. The function of the reporting database is two-fold. Firstly, it serves as a place where we can store company-specific information that does not need to clutter the data models of the blockchain. The company-specific information will also be easily erasable in case a partner leaves the platform. Secondly, it serves as a much faster way for querying information instead of calling the blockchain nodes every time we

[1] A programming language that is one of the core technologies of the World Wide Web,

[2] JavaScript framework for building user interfaces and single-page applications.

need some information. A choice was made in favour of a relational DBMS for development purposes but in an industry-grade application ready for deployment, a NoSQL DBMS will be more appropriate where the data will be replicated, sharded[3] and distributed to the validating nodes of the company that owns the data.

3.1.4 Modular Design

The back-end systems that make everything possible need to be highly modular and be able to be deployed and run on a variety of computers without extensive configuration of the specific environment. For that very reason, we need to enforce the separation of the back-end systems and build it using a micro-service architecture where different components will interact with each other through prespecified channels of communication. For the aforementioned reasons we need to build the components into Docker[4] containers because they are very flexible and allows deployment of applications in separate containers independently and in different languages. It also reduces the risk of conflict between languages, libraries or frameworks when they are containerized.

3.1.5 Client vs Server Signing Model

When using a REST API there is a consideration to be made of what signing model is the most appropriate for the use case. Like everything else in computer science, both models have their caveats.

Client signing model

The workflow of using a client signing model requires the client to be able to build (batches of) transactions and sign them using the user's private key. Then the client should serialize the batches and send them to a single POST endpoint to the REST API which will directly send it to the validator. The REST API in this case is only a mediator between the client and the validator.

The advantage of this model is that transactions are signed locally using the user's private key and thus their identity is fully verifiable. Even in the event that the server is fully compromised, there is no way to falsify transactions.

The disadvantage of this model is the rest api only contains 1 endpoint for submitting transactions and thus does not follow RESTful standards instead of supporting the full CRUD[5] array of manipulations for a given entity. Another disadvantage is that every client needs to implement transaction creation, serialization and signing functionality.

This model leaves the responsibility to the users to protect their own private key and requires the use of rich clients[6]. Sawtooth provides a default REST API which only submits transactions which are already serialized. However, this is not the signing model this project follows and thus a custom REST API is needed.

[3] Sharding is a method for distributing data across multiple machines[21]

[4] Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.

[5]

[6] computer that typically provides rich functionality with little to no dependence on the central server.

Server signing model

The Server signing model abstracts the transaction signing from the clients by putting it in the REST API. The client submits update requests as JSON objects to traditional RESTful endpoints. Then the server creates and signs transactions based on the submitted JSON and also handles batching and serialization. At the end, it sends the transaction to the validator.

The advantage of this approach is that the server maintains a RESTful interface, which means that the interaction with the server is no different than if the server was backed by a traditional database. Consequently, this simplifies the clients because they don't have to handle transaction creation, serialization, deserialization and signing.

The disadvantage of this approach comes from the need for a server to sign transactions which somehow compromises the identity verification of a blockchain. A naïve approach of this signing model would be having a single private key for signing all transactions. This will negate all the identity verification advantages of blockchain because the server would be the single source of truth, the same as with a traditional database.

A more robust solution can be achieved by generating a public/private key pair for each user on the server and storing the (encrypted) private key in a traditional database (ideally an additional key escrow service is recommended but this project does not implement this additional security measure). The server can then retrieve and decrypt the private key of an authenticated user and sign transactions on their behalf. While this is better than the naïve approach, it is only as secure as the server's security mechanisms. The Metal Supply application uses this approach.

3.2 Metal Supply Architecture Overview

The Metal Supply application includes these components:

- **Transaction Processor**. The transaction processor defines the business logic and validates transactions. It is created using the Sawtooth Python SDK and thus it has built-in interfaces for simple communication with the Sawtooth Validator. Together the Transaction Processor with the Sawtooth Validator make the Validator Node.

- **Web Client**. The web client is a lightweight single page application written using the JavaScript framework Vue.js. It has the capability to authenticate and authorize agents on the platform, submit transactions to register new agents, add new records, add record locations, transfer ownership of records, etc. Transactions are submitted as HTTP requests with JSON bodies to the REST API.

- **Rest API**. The REST API handles communication with the Validator node. It creates batches of transactions according to the requests submitted from the web client. It manages the public/private keys, kept in a reporting database, and signs transactions with them. Also queries the reporting database when a request for retrieval of information is received from the Web Client.

- **Event Subscriber**. The event subscriber handles all the delta changes in the state data. It subscribes to the Validator Node in order to receive the state data changes as events and in turn update the reporting database.

3.3 Application Design

Before going into the specific components, it is essential to go through the application design.

In the Metal Supply application records are the metal products that are tracked throughout the supply chain, whereas agents are the users of the system, or in other words, the agent is the supply chain manager. The Transaction payload is the set of fields provided to the Transaction Processor in order to update the state.

3.3.1 GDPR Compliance

There have been considerations about adding additional information about the agent/record in the blockchain state. Information such as name, phone, company, etc. This is the intuitive solution when designing a normal database schema. However, blockchain has an additional property that makes it unique - immutability. Consider the situation when an agent has been using the system for years but decides to leave the platform. Under Article 17 the data subject shall have the right to obtain from the controller the erasure of personal data concerning him or her[22]. Hence, we are required to delete everything that can be used to identify directly or indirectly someone on the blockchain. This consideration led to the offloading of personal information to an off-chain database which is also used for reporting purposes. That way records on the blockchain could not be linked to real users. In the case of a request for the erasure of personal data, we could easily delete the rows in the reporting database about the given user.

3.3.2 State Data Models

Payload Serialization and Deserialization

The validator sees the data as simply a string of bytes. Only the transaction processor that handles the transaction will deserialize the payload. When interacting with the blockchain, a transaction processor sets and retrieves state data by making calls against a version of state at a specific address that the validator provides.

Likewise, when a client sends a transaction to the validator, it must serialize the payload data. For this reason, the encoding scheme must be deterministic; serialization and deserialization must always produce the exact same results.

Consequently, all data models in the Metal Supply application are defined using protocol buffers[7] (protobufs) to encode all objects before storing them in the state (both payloads and state data). Metal Supply uses protobufs to serialize batches and transactions because they serialize and deserialize identically for the purposes of the project.

Agent Data Model

Each agent has a public and private key associated with them. Keep in mind that private keys are not stored in the state(the blockchain), instead it is encrypted and stored in an off-chain database. The agent can be 1 of 3 types - Waste Owner, Recycler, and Converter. A Waste Owner is in a possession of end-of-life metal products. Recyclers are the ones separating

[7]Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data

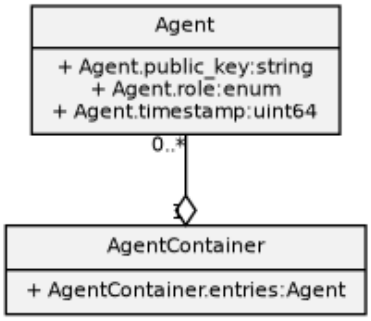

Figure 3.1: Agent Data Model

all the different metals inside the metal products. Converters take the separated metals and convert them into new products. An agent can create, transfer and update records. The Agent data model is defined in Figure 3.1.

Record Data Model

A record consists of many parameters that define the record. See Figure 3.2. A detailed explanation of the fields can be found below:

- **Record ID** The ID of the record
- **List of Next Record IDs** The record ids of all records that were made as a result of this record
- **List of Record Owners** A List of all the owners of a given record. Each Record Owner has the following fields:
  - **Agent ID** - The public key of the agent
  - **Percentage Owner** - The percentage of ownership of the record by the agent
  - **Timestamp** - Timestamp of an agent becoming an owner of the record
- **Material Type** The type of material
- **Material Origin** The origin where the material come from.
- **Tonnes** The weight of the record in tonnes
- **List of Record Contents** A list of metal contents. Each metal content consists of a metal and the percentage of that metal in the record.
- **List of Record Locations** - All locations of a given record. A Location consists of:
  - **Latitude and Longitude** - The geographical coordinates of the record
  - **Agent ID** - Public key of the agent that owns the location because a record could have more than 1 owner.
  - **Timestamp** - Timestamp of the when the location was recorded.

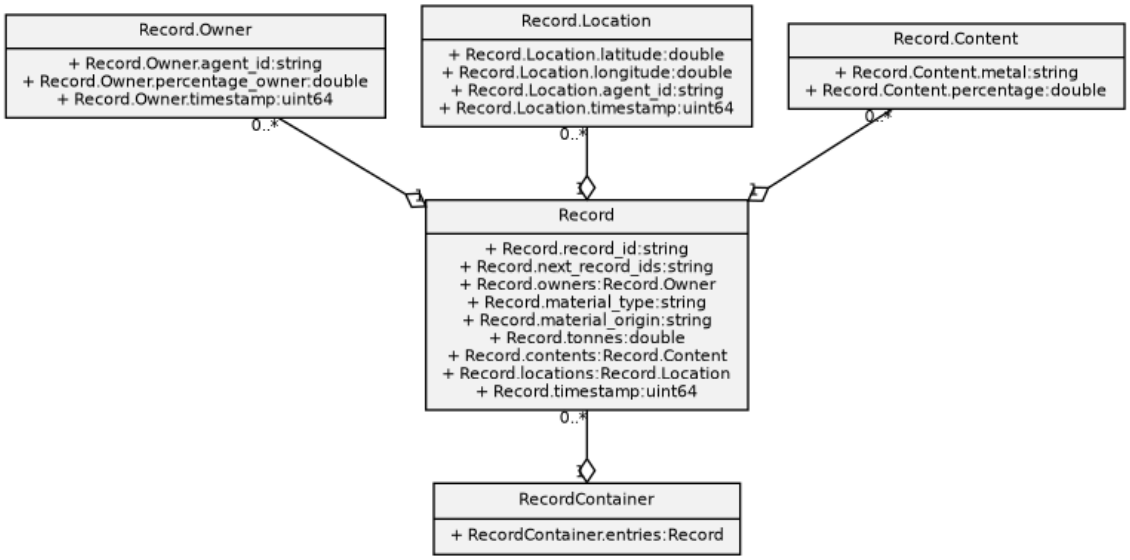

Figure 3.2: Record Data Model

Payload Data Model

Another essential data model that is required by the Transaction Processor is the Transaction Payload. Do note that the signature of the agent(owner) is not part of the payload but it is received by the transaction processor as well. We have 5 types of payloads:

- **Create Agent** This payload only requires the agent's name and role. The agent's public/private keys are generated by the Transaction processor and send as an event to the subscriber.
- **Create Record** - This payload requires the record id which is generated by the rest api, the geographical coordinates, material type and origin, weight in tonnes and the contents of the record
- **Update Record Location** - The payload requires the record id for identifying the record, the geographical coordinates and the agent id that owns the location. The transaction will not be valid if the signer is not one of the owners of the record.
- **Transfer Record** - The payload requires the record id for identifying the record to be transferred, the public key of the receiving agent and a percentage to be transferred. The transaction is not valid if:
  - the signer is not one of the owners of the record or does not exist
  - the record does not exists
  - the receiving agent does not exist
- **Link Record** - The payload requires the two record ids of the records that will be linked. The transaction is not valid if:
  - either of the record ids is a non-existing record.
  - the signer is not one of the owners of the record.

3.3.3 Database Schema

Before building the components, a database schema should be defined. See Figure 3.3. The reporting database has three types of columns: a blocks table, on-chain tables, and off-chain tables. The blocks table represents all block numbers and hash ids. The on-chain tables represent the data stored on the blockchain and analogically the off-chain tables - the data not stored on the blockchain. All updates to the blocks and on-chain tables should happen atomically, within a single database transaction. This ensures that the state of the blockchain remains consistent with the database data.

Starting from the blocks table, it is meant to represent the history of block numbers and hash IDs in the blockchain. The table is used for fork resolution to compare the block number of a received event against its block ID. If a new event comes with a block number and a block hash id that is in the blocks table, that means it is a duplicate block and we should ignore it. In the case that only the block number is in the blocks table but the block hash id is different, that means a fork is detected and we should delete all entries in the database which are greater or equal to the block number and replace them with the new block data. Finally, when none of them is in the blocks table, the state change is applied throughout the application-specific tables.

Each table that represents on-chain data has a start and end block number columns. The start block number identifies the block number where the data is taken from. The end block number defines whether this is the most recent version of the data (value of max integer) or if the data has been superseded by another block (value of the newer block number).

The agents, agent_role and auth tables represents the agent data model on the blockchain. The agent_role table has a one-to-one mapping with the agents table and represents the three role ids and the corresponding name of the role - Waster Owner, Recycler and Converter. The auth table stores off-chain data and has a one-to-one mapping with the agents. The table also contains the information needed for authentication: public key, hashed password and encrypted private key. The agents table includes both on-chain columns (e.g. public_key, role, timestamp) and off-chain columns(name, email, company, hashed_password).

The parent table records and child tables record_owners, record_locations, record_contents, record_links represent the record data model on the blockchain. Every column in those tables represents on-chain data except for the published column, in some of the tables, which represents whether the records are marketed on the website. All child tables have a many-to-one relationship with the parent table records because a single record can have multiple locations, owners, contents and links.

3.4 Choice of Consensus

Throughout the whole development and testing process, the Dev mode consensus algorithm was used, provided by Sawtooth for exactly that purpose. It selects a random leader that writes the transaction to the blockchain. After the system goes into production, there are many variables that need to be taken into consideration. For example, if the network is going to be used by a small number of organisations in a consortium but not less than four, PBFT will provide the best performance. On the other hand, if the network is used by a large consortium of organisations and/or open membership is required, the best consensus will be PoET. However, in the same case, if the hardware on which the blockchain is going to be run does not support a special CPU instruction set called Intel Software Guard Extensions (SGX), then the PoET

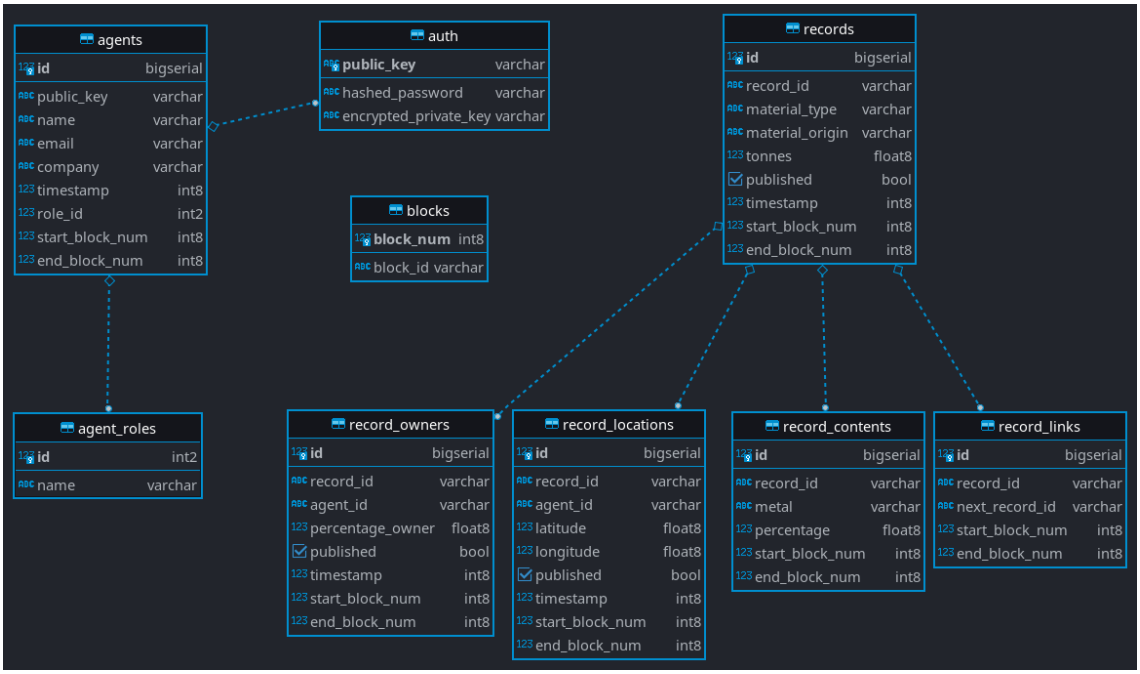

Figure 3.3: Reporting Database Schema

simulator is going to be the obvious choice. The great thing about the platform is that we can swap the consensus algorithm dynamically even after the network goes into production. This enables us to choose the best consensus algorithm depending on the maturity of the platform. If the network is used by one company with a small number of nodes, maybe the dev mode consensus is not a bad choice. After the nodes grow more than four or more organisations join, we could switch to PBFT. When the network becomes even bigger, we could test how PoET performs and choose a version supported by the available hardware.

3.5 Chapter Summary

This chapter describes the design requirements and the needed planning involved in the project. It also provides detailed description of the data models used. The chapter also covers the architecture of the application, as well as choices made in the process of planning and development.

Chapter 4

Implementation

4.1 Disclaimer

The project is built on top of an example project created by the Hyperledger Sawtooth team demonstrating how to use the enterprise blockchain platform Hyperledger Sawtooth and providing most of the boilerplate code for connecting all components together. The Sawtooth Simple Supply software is licensed under a Creative Commons Attribution 4.0 International License and I have given appropriate credit where it is due, as well as indicating that I changed it according to the creative commons license. The components I built my project on top of are the Transaction Processor, Rest API and Event Subscriber by extending them to support the functionality defined by the application-specific business logic.

4.2 Architectural Components

The architecture of the system can be observed in Figure 4.1

4.2.1 Transaction Processor

When the Validator receives a new transaction, the validator verifies the signature, and then sends the transaction to the Transaction Processor for validation using the business logic. The transaction processor has a few important functions. Firstly, it decodes the payload and makes sure it is properly formatted. Then it needs to validate the transaction using the applications-specific business logic.

In order to perform validation, the state data needs to be retrieved. As with the payload, Metal Supply needs to ensure it fetches the right data at a given address. Because both the client and transaction processor need to generate addresses for state entities, the address-generation functionality is encapsulated in its own small library. The library defines the methods for creating an address based on the identifier of a state entity. To generate an agent address we use:

- the first 6 characters from the SHA-512 hash of the namespace which is ”metal_supply”
- the entity prefix - ”00” for agent and ”01” for record
- the first 62 characters from the SHA-512 hash of the entity identifier

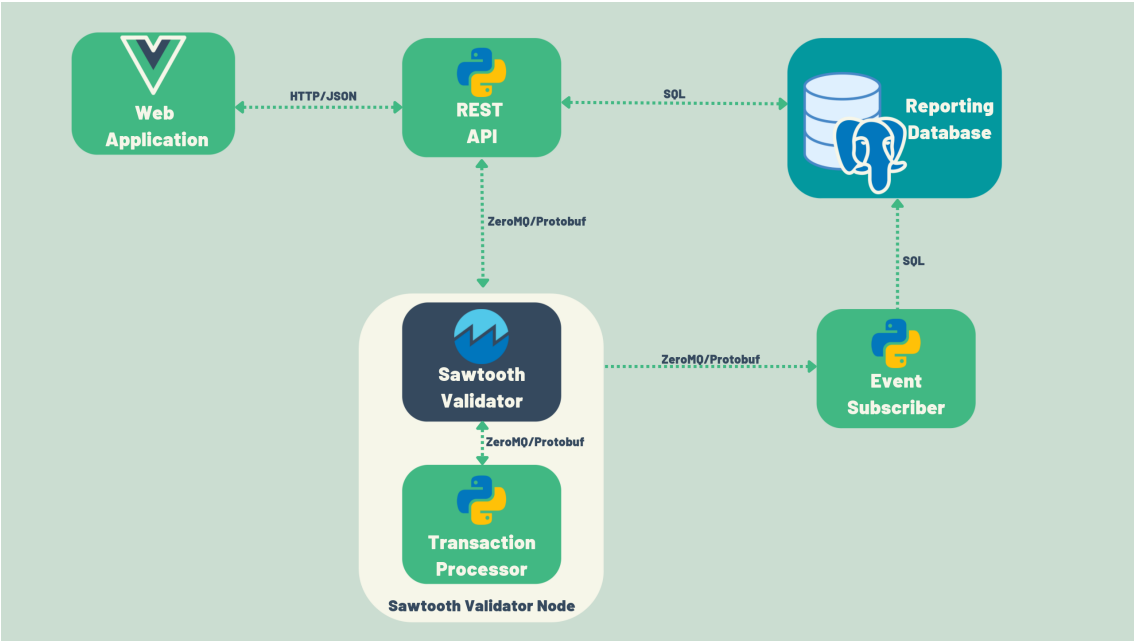

Figure 4.1: Application Architecture

Because of the fact that all state entities are stored in containers, when updating the state, the transaction processor must either create a new container or reuse an existing one at a given address.

The Transaction Processor is written in Python because the Sawtooth SDK in Python is the most mature and feature-rich. The SDK also provides an easy and intuitive API for communication with the validator using ZMQ/Protobuf and different channels to handle requests concurrently. In Figure 4.2 we could observe a simple piece of business logic handling the record transfer implemented using the Sawtooth SDK. Initially, the address is taken using the small addressing library defined earlier. Then the container at this address is taken from the state using the application context. The container's content is deserialised into Protobuf objects which enables us to work with native python objects. We look through all entities in the container until we find the one we need. Then we write our business logic using generic python code. It transfers a given percentage of ownership between two agents. For example, if Agent A and B both have 50% of a record and Agent A transfers 20% of the record to Agent C, then Agent B should stay with 50%, Agent A's ownership is decreased to 40% and a new owner is introduced to the record's array of owners with 10% ownership. Naturally, if the agent is not within the list of owners, the agent is added. The container with the new data is serialized again and written to the context at a given address.

4.2.2 REST API

The Simple Supply REST API handles transaction creation and signing logic. This custom REST API has several routes for submitting transaction requests and fetching data for use by a user-facing client application. Metal Supply includes a single page application web app that uses these routes but the routes can also be accessed by making simple http requests.

The REST API is implemented using the aiohttp library. It is a lightweight and easy-to-use

```python
    def transfer_record(self, receiving_agent, sending_agent, record_id, timestamp, percentage):

        address = addresser.get_record_address(record_id)
        container = record_pb2.RecordContainer()
        record = record_pb2.Record()
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container.ParseFromString(state_entries[0].data)
            for record_entry in container.entries:
                if record_entry.record_id == record_id:
                    record = record_entry

        sending_owner = next(owner for owner in record.owners if owner.agent_id == sending_agent)
        receiving_owner = next((owner for owner in record.owners if owner.agent_id == receiving_agent), None)
        receiving_percentage = percentage * sending_owner.percentage_owner / 100.0
        sending_owner.percentage_owner -= receiving_percentage

        if receiving_owner is None :
            owner = record_pb2.Record.Owner(
                agent_id=receiving_agent,
                timestamp=timestamp,
                percentage_owner=receiving_percentage)
            record.owners.extend([owner])
        else:
            receiving_owner.percentage_owner += receiving_percentage


        data = container.SerializeToString()
        updated_state = {}
        updated_state[address] = data
        self._context.set_state(updated_state, timeout=self._timeout)

```

Figure 4.2: Transfer Record Code Snippet

python library for defining a simple server that is using RESTful interfaces. The REST API handles incoming HTTP requests, handling authentication, fetching requested data from the Database, and building and sending transactions to the Validator Node.

The basic workflow for creating an agent is as follows:

- The front end sends a new HTTP POST request with a JSON body(email, password, role) to the ”/agents” route in the REST API.
- Check all required fields within the body are present and validate fields
- Generate a new public/private key pair for the agent
- Encrypt the private key and store it in the auth table in the reporting database, along with the agent’s public key, hashed password and other personal data.
- Format the transaction and send it to the validator
- Given all operations are successful, the REST API returns an auth token to the front end which could be used for authentication the user when calling protected routes that only authenticated users can access. One example of such a protected route is the creation of a record because it requires an agent attached to it as an owner.

In order to confirm user identity, the transaction needs to be signed with the private key of the agent before the transaction is sent to the validator. To achieve it each user needs a 256-

bit key. For signing, Sawtooth uses EDCA standard algorithm with secp256k1 curve[1] Using the server-signing model, is the only way a user can prove their identity on the blockchain. Hence, the private key must be kept secret and secure because anyone can sign transactions on the user's behalf given access to the private key. That is the reason why the private key is not stored as text in the reporting database but it is encrypted using AES[2]. The way the private key is encrypted is by using a secret key defined in the application. The secret key should be ideally defined using an environment variable or taken from a secure secret key repository. In this project, it is defined in the application for simplicity. In order to ensure that the same value encrypted multiple times, even with the same secret key, will not always result in the same encrypted value, we should use an Initialization Vector (IV) defined in the AES specification[24]. This is usually a randomly generated sequence of characters but in this case, the agent's public key is used as an IV instead. In order to decrypt the encrypted private key, we use the same secret key defined in the application and the IV - the agent's public key.

When we receive a request for creating a new agent, the personal information is stored directly in the reporting database. Each entry in the reporting database has a start and end block numbers which are updated by the Subscriber when a block is written to the database. That is why initially those columns in the given entity database table are set to null. After the transaction is submitted and a new block is written on the blockchain, the Subscriber receives an event which populates those fields with the appropriate block numbers. This has been done with the intention that an entity should not be considered existing just because its private data is on the reporting database. It has to be first written on the blockchain in order to be considered a real entity. Consequently, when agents or records are retrieved from the reporting database in response to an HTTP GET request on the given entity, the start and end block numbers are always null checked.

After private data saving and public/private keys generation, we need to format the transaction, serialize it and send it to the validator. However, we firstly generate a transaction signer with the private key. Then we create a single transaction wrapped in a batch, and then send the batch to the validator. In order to create a transaction, we need to define the state address where a transaction processor is allowed to read/write during the execution of the transaction. In the case of transferring a record, the transaction processor should be able to read from the sending and receiving agent addresses and the record address. To generate the addresses we use the mini-library for addressing we defined earlier. After that, we create the payload using the Protobuf Payload object and serialize it to bytes. After that a transaction is created, it is being wrapped in a batch and sent to the validator. The batch can be referenced by the block header signature, which allows us to query its status.

4.2.3 Event Subscriber

Reading data from the blockchain consists of finding the address where the entity of interest lives and going through all entities in the given container until you find the one that is needed. Although this data can be fetched directly from the validator fairly easy, it does not scale well in terms of speed when the number of entries rises in the magnitude of millions. That's why

[1] secp256k1 refers to the parameters of the elliptic curve defined in Standards for Efficient Cryptography (SEC) [23]. It is also used in Bitcoin's public-key cryptography.

[2] The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information[24]. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext[24].

an event subscriber and a reporting database are set in place to make fetching data as easy and fast as possible for the clients. The Event Subscriber subscribes to Sawtooth state delta and blocks commit events which are sent when the validator's state database is updated. The event contains state data which is processed by the Event Subscriber and propagated to the reporting database which becomes essentially an off-chain copy of the database. The reporting database allows for rapid access to information when the REST API queries the state of the blockchain.

The workflow is as follows:

- The event subscriber establishes a connection with the validator using the Sawtooth SDK
- The event subscriber specifies the event type that it wants to receive from the validator. In this case, both "block-commit" and "state-delta" are needed.
- The event subscriber also specifies all known block IDs and everything is sent to the validator using the Sawtooth SDK. For the first event subscription, a null block id is sent in order to request state updates from the genesis block.
- After subscribing, the event subscriber listens for incoming events
- When the client receives an event, it checks whether the event data represents a duplicate event or a fork using the block id.
  - If the block is duplicate, we ignore it
  - If the block represents a fork, the event subscriber updates the reporting database by transforming the raw state data into application-specific database table rows.
- In the event of a possible fork, all entities that were changed as part of the fork are updated by being set to "ended", whereas the newly inserted ones are marked as "started".

The "started" and "ended" flags are represented with the columns "start_block_number" and "end_block_number". For simplicity, I will refer to the block number of the block that contained the batch creating/updating the agent/record with the "current block number". When a row representing an agent/role is inserted with private data (data that is not on the blockchain), both of those columns are set to null. When an agent/record is received as an event:

- the "start_block_number" is set to the current block number.
- the "end_block_number" is set to MAX_INTEGER indicating it has "started".

When a consecutive event is received about an already existing agent/record, we "end" the old rows and insert the new row with the updated data in the following way:

- the "end_block_number" of the old row is updated to the current block number indicating it has "ended".
- the "start_block_number" of the newly inserted row is set to the current block number.
- the "end_block_number" of the newly inserted row is set to the max block number indicating it has "started".

This has been done because the database might contain the same object in state at different block heights.

Fault tolerance was one of the main considerations when the event subscription technology was developed by Sawtooth. In case the Event Subscriber crashes, when it is restarted, it will be able to send all known ids from the database and receive a ”catch-up” event that contains all blocks since the Event Subscriber went offline. On the other hand, if the on-chain data in the reporting database is partially deleted for some reason, by supplying what is left of the known block ids, the data will be restored.

4.2.4 User Interface

The user interface is built using Vue.js. The intention behind it is to build a web application which will abstract away the use of blockchain on the back-end for normal users. Each user logs in with an email and password instead of public and private keys (which are also a valid form of authentication). That way the end-users will receive all benefits of a blockchain system without dealing with the more obscure elements that come with it. The web application has the following features:

- Authentication with JWT[3]
- Form data validation before submitting the data to the REST API
- Navigation Bar changes and page restrictions based on logged-in status.
- Public Agents and Records pages with pie charts for metal contents
- Profile with user’s records in detail along with a historical analysis of the record.
- Google Maps integration for the display of locations and flow of dependant records.

User Registration and User Login

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties[25]. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS), enabling the claims can be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted[25]. The exact way the JWT is created is by generating a signature of the public key of the user with a secret key held in the application using the JSON Web Signature specification. That way we could ensure the JWT has been issued by the server which has already authenticated the user.

In Figure 4.3 we can see a sample flow of registration, login and then accessing protected routes with a JWT. To begin from the start, in order to register a new user the web app supplies their email, password(base64 encoded) and role on the POST endpoint ”/agents”. Then the back-end verifies the email and password and returns the authentication token (JWT) and the public/private key pair generated for the user. The login flow is very similar, we send the email and password(base64 encoded) to the ”/authentication” endpoint which returns the authentication token (JWT) only. This token, along with all user data (excluding the public/private key pair) is saved in Browser Local Storage. In order to retrieve and write to the Local Storage, we use Vuex which is a state management centralized store for all the components in

[3]JSON Web Token

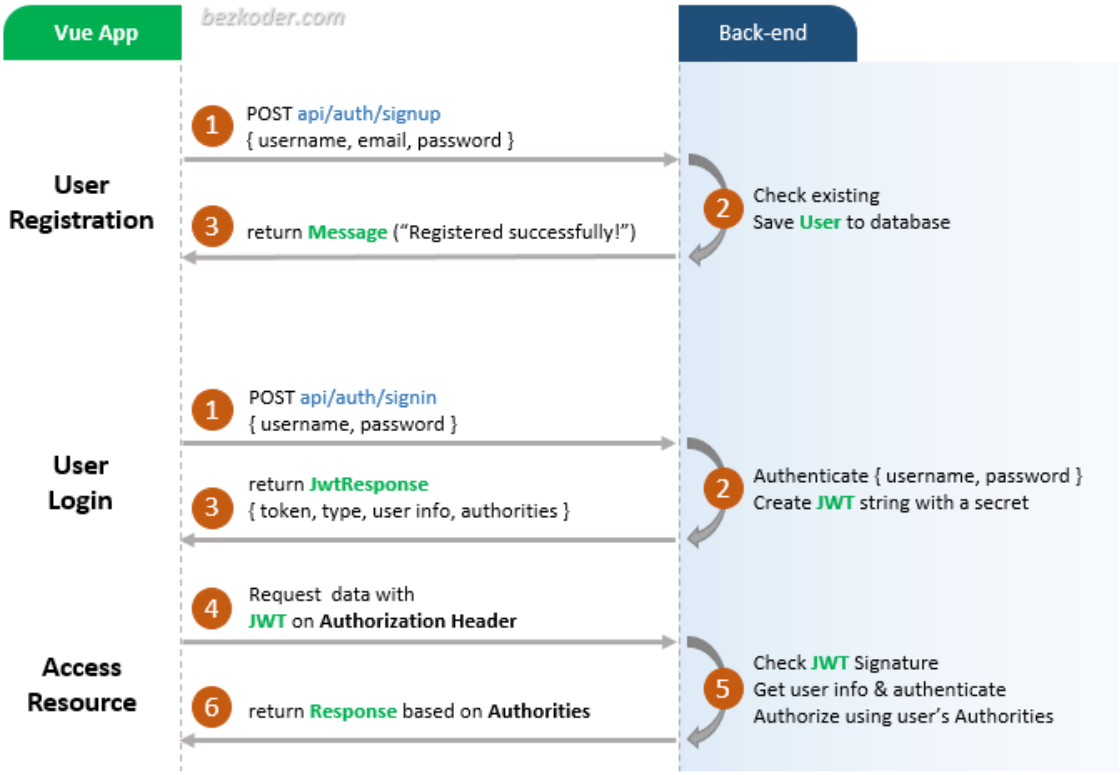

Figure 4.3: Vue.js Authentication with Vuex[4]

an application[26]. Using it we could retrieve information about the user in any component. When a protected route is requested, an Authentication header is added to the request along with the JWT to authenticate the user. Otherwise, the REST API will return HTTP status code 401 Unauthorised.

Web Application Structure

The structure of the Vue app is highly modular. See Figure 4.4. Each component is built with the intention of being reused as part of another component. Take for example the *Content Chart* component which displays a pie chart of the different metals in a metal product. It has been used on the *Records* page where all records are shown, *My Record* page where a detailed analysis of a record is displayed and the *Profile* page where all user's records are shown.

There is also a separation between the components and services. Components contain HTML, scripting and styling elements. All services used for calling REST APIs are abstracted away from the scripting part of Vue components in order to be reused in more than one component.

The router defines all routes of the web application and their corresponding component that needs rendering. It also defines the routes which are protected and can be accessed by authenticated users only. It can also define route rules based on the role of the user in case Waste Owners, Recyclers and Converters need to see different web content for a given reason.

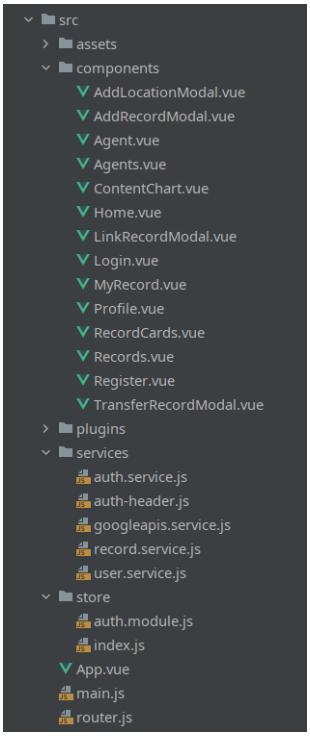

Figure 4.4: Web Application Structure

Google Maps Integration

In order to integrate with Google Maps, firstly an API key needs to be issued by Google in order to track usage for billing purposes. Using this key we can make queries to the Google Maps API. The Google Maps API is used primarily for displaying a map of the world with markers and lines indicating all the locations of a metal product and the path of all materials that were used in making it. It is also used for converting geographical coordinates into country and district and vice versa for better interpretation of the locations by end-users.

4.3 Application Endpoints and Workflows

There are eleven endpoints in the REST API which define the functionality of the project. They are split into two parts. Endpoints for agents and records. This section will provide a discussion on the functionality provided by them.

4.3.1 Endpoints

The endpoints are the following:

- Agents endpoints:
  - **POST /authentication** - Authenticates the user given email and password. The password is checked against the encoded password in the reporting database without making any interactions with the blockchain.

- **POST /agents** - Create a new agent given both personally identifiable information and generic one. This endpoint interacts with both the database and the blockchain to create the agent. Does not rely on authentication.
- **GET /agents** - Get all public agents on the platform. Does not rely on authentication.
- **GET /agents/records** - Fetch all records owned by the given agent. It requires authentication and it only queries the reporting database.
- **GET /agents/{agent_id}** - Fetch the data about a specific agent. No authentication needed and it only queries the reporting database.

• Records endpoints:

- **POST /records** - Create a record given information stored in both the blockchain and the reporting database. This measure is not taken for personality revealing information but rather for the separation of relevant data. It requires an authenticated user and both interactions with the reporting database and the blockchain network.
- **GET /records** - Get all public records on the platform. Does not require authentication and only queries the reporting database.
- **GET /records/{record_id}** - Fetch all the information about the record from the reporting database. It requires authentication.
- **PUT /records/{record_id}/transfer** - Transfer a certain percent of the ownership to another agent. It requires interaction with the blockchain only and also requires authentication.
- **PUT /records/{record_id}/location** - Add a location to an existing record. It requires interaction with the blockchain only and also requires authentication.
- **PUT /records/{record_id}/link/{next_record_id}** - Make a link between two records. It requires interaction with the blockchain only and also requires authentication.

4.3.2 Workflows

From the aforementioned endpoints, we can group them into 3 categories based on the workflow they follow.

The retrieve workflow is the most simple and intuitive workflow for people used to traditional database-backed applications. See Figure 4.5. All retrieve operations follow this flow. The request is made by the web application using HTTP/JSON to the REST API. It queries the reporting database using SQL and returns the retrieved data as a response to the web application.

In the create workflow the blockchain network is also included. See Figure 4.6 When the web application makes a request to create an agent/record, the REST API queries and writes database-specific information relevant to the request. Afterwards, the REST API creates a transaction and sends it as raw data to the Sawtooth Validator Node using the ZeroMQ protocol. The Sawtooth Validator verifies the block header (including the signature) and redirects the raw data to the Transaction Processor. The Transaction Processor parses the raw data into the predefined Protobuf objects and validates it against the business logic. When the transaction is submitted as a candidate block to the other validator nodes, the REST API receives a response

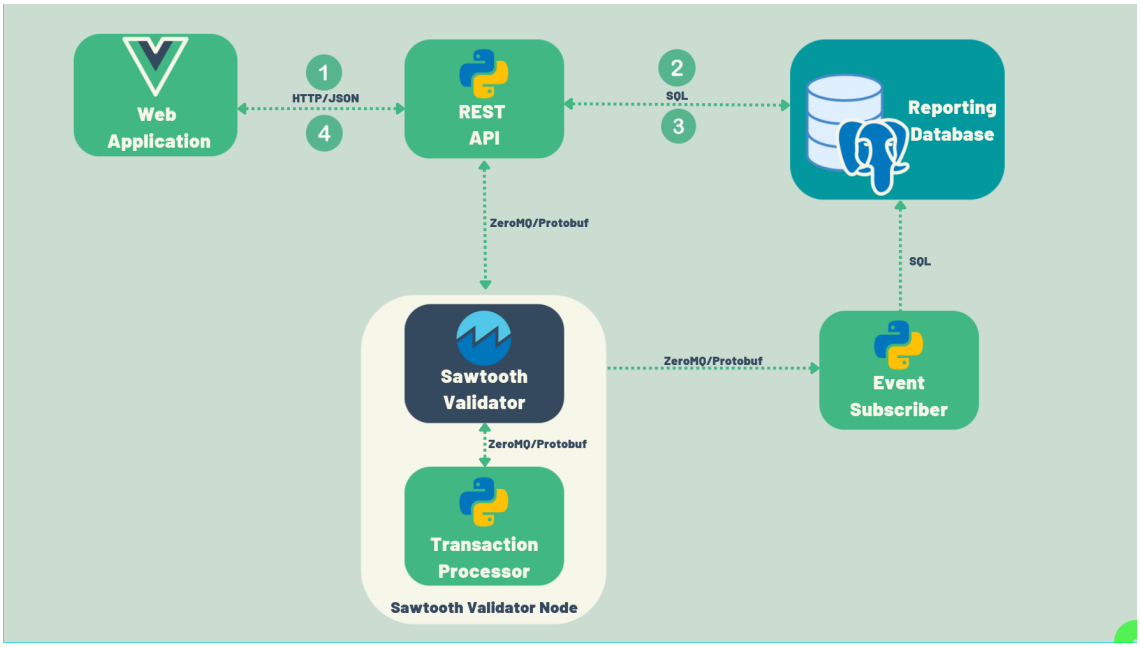

Figure 4.5: Retrieve Workflow

which is redirected to the web application that the transaction has been submitted. Almost asynchronously to the response, the Event Subscriber receives an event that a new block has been written on the blockchain and after parsing the raw data from the block, it writes it into application-specific database tables in the reporting database.

The update workflow is very similar to the update flow but there is no interaction with the database by the REST API See Figure 4.7. The database is updated by the Event Subscriber when the event is propagated through the Sawtooth Network and written on the blockchain. That way, we keep the on-chain and off-chain states in sync in case the transaction is rejected by the Transaction Processor.

4.4 Docker Containerization of Services

The system was built from the ground with the intention of being fault-tolerant, environment agnostic, lightweight and secure. By wrapping the services in docker containers we could check all those boxes. Docker containers could be run on any platform without additional configuration of the environment. What is more, if a service crashes for a given reason, Docker can be configured to restart that service. If there are dependencies associated with the service (See Figure 4.8), docker can start/restart them in the desired sequence. For example, the consensus algorithm engine run in a container depends on a validator. The user interface also depends on the REST API. Finally, using docker containers improves security because every service is run in an isolated environment. Hence, even if one of the services is exploited, the breach of security will be minimized to that service only.

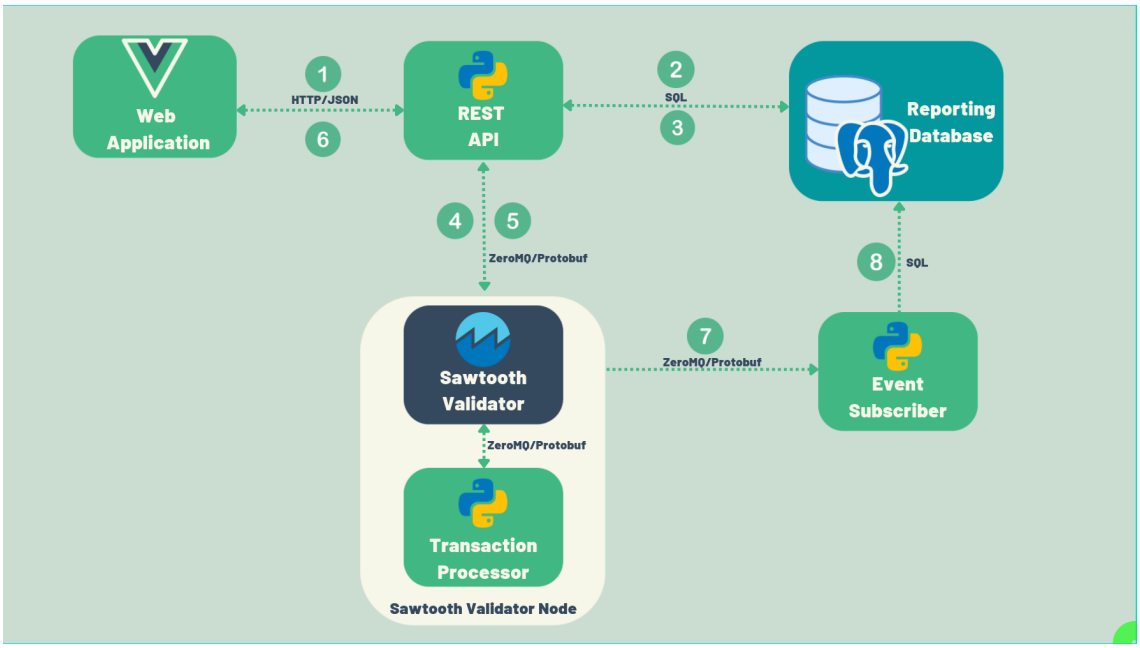

Figure 4.6: Create Workflow

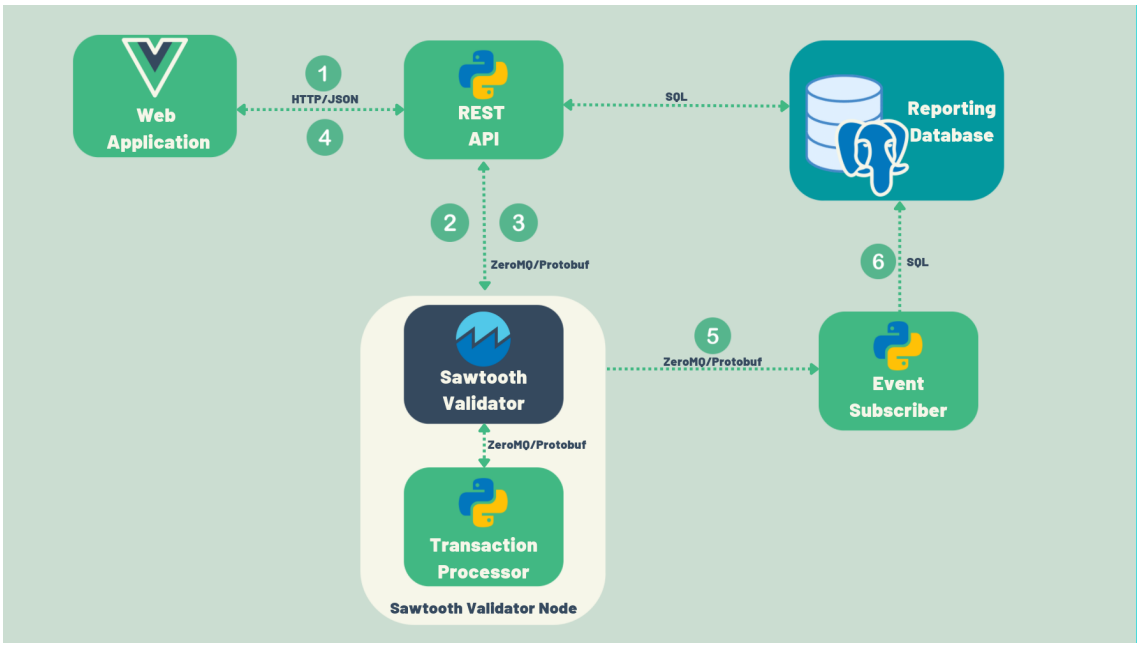

Figure 4.7: Update Workflow

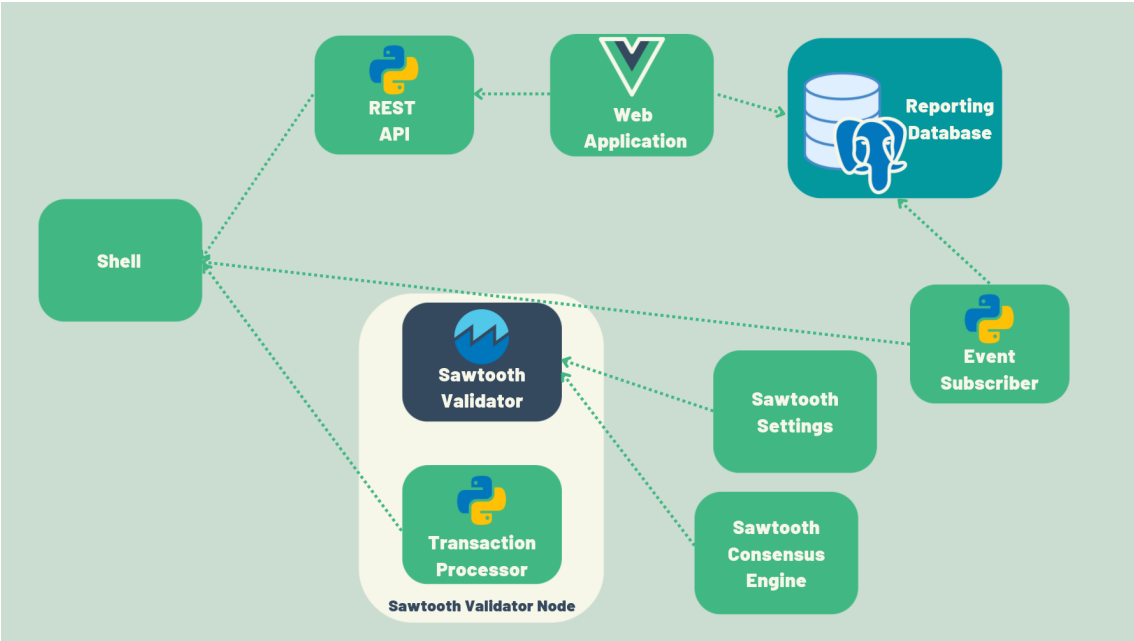

Figure 4.8: Component Dependencies

4.5 Directed Graphs of Locations

When displaying the analysis of a metal product, the application should also display all metal products included in the manufacturing of the current one. Of course, for better visualizations, the front-end should arrange all of them on the map and connect them with directed lines to indicate where they came from. In other words, the application should display a directed graph. See Figure A.7

To begin with, the database holds all record links as two columns in the table record_links - the record id and the connected record id in the other column. See Figure 3.3. The REST API should construct an adjacency list out of the entries in this table to allow the front end to display it properly. However, a full adjacency list is not needed. We can traverse this table to find only the records linked directly or indirectly to the current one. The REST API retrieves all records ids for which there is a next record id equal to the current record id. Then, it retrieves all the data about them and starts the same process recursively for every connected record. The REST API looks records up in reverse order because linking is done from input material to output material, whereas we want to get all input materials for a given output material. Using this approach, the REST API should retrieve all records linked directly and indirectly to the current one. The base case of the recursion will be reached when a given record does not exist in the next record id column of the record_links table. Do note if we are encouraging a circular economy we should expect cycles in the graph. To deal with this inconvenience from a technical point of view, we should keep a list of visited records. This list will be extended every time the full information about a given record is retrieved by adding it to the list. If any of the linked records is already in the visited records lists, we should simply ignore it as that means we have already explored it. The aforementioned algorithm is using a breath-first search in order to construct an adjacency list.

4.6 Chapter Summary

This chapter outlined the four main architectural components of the system in details. It also discussed why components are build into containers and how locations have been turned into directed graphs.

Chapter 5

Evaluation and Testing

5.1 Performance Evaluation

The performance of the project was evaluated with two criteria. Load testing how many requests it can handle per minute and testing speed of retrieval of information directly from the blockchain and from the reporting database.

5.1.1 Reporting Database and Blockchain Performance

Each test was performed ten times and the average and standard deviation of the response times were calculated.

Record Retrieval

The test in Table 5.1 demonstrates how fast the reporting database is for retrieving records. The reporting database achieved an average response time of sub ten ms with a very small deviation from test to test. On the other hand, the Blockchain performed very slow for retrieving records compared to the previous result. It demonstrated an average response time of around 450 ms with a standard deviation in the same neighbourhood. Those results reveal that the Blockchain was very inconsistent in the tests.

Source	Average (ms)	Std. Dev (ms)
Reporting Database	9.1	1.66
Blockchain	447.36	306.39

Table 5.1: Record Retrieval

Agent Retrieval

The test in Table 5.2 once again demonstrated a clear advantage for the reporting database regarding the speed of retrieval. In this test, we can observe that the reporting database was slightly faster compared to the database retrieval of records. At the same time, the blockchain was noticeably slower, resulting in a more than one hundred times speedup of the reporting database, compared to retrieval from the blockchain.

Source	Average (ms)	Std. Dev (ms)
Reporting Database	4.7	1.05
Blockchain	618.9	243.52

Table 5.2: Agent Retrieval

5.1.2 Load Testing

The load test was performed using the industry-grade load testing tool JMeter. Each of the load tests below was run with 10 threads and a maximum of 1000 loops per request per thread. If the throughput was low, the load test was stopped before reaching the limit when the results converged. The load tests were performed with around 50 existing agents and records. The Average, Min and Max column names refer to response time in ms. Each services in the project is run in a multi-threaded environment which is able to take advantage of a full core clocked at 4.0GHz.

Retrieve Records and Agents

Judging from Table 5.3 when retrieving data from the reporting database directly we can observe a noticeable difference in average response time between 2 types of requests - getting agents and records. Agents seem to be retrieved very efficiently with an average response time of under 10ms. Getting all records, on the other hand, had a response time of just above 200ms. This could be explained by the fact that records are split into many database tables with a one-to-many relationship, whereas agents only have related database tables with one-to-one mapping in addition to having much fewer columns to retrieve. During the load test, 2 of the components reached their maximum utilization - the REST API and the Postgres DB with the utilization of 98.8% and 99.6%, respectively. The rest of the components did not increase their utilization.

Request	Total Samples	Average (ms)	Min	Max	Std. Dev	Throughput
Get all Agents	10000	8	2	30	1.62	36.6/sec
Get All Records	10000	217	44	344	26.98	36.5/sec
Get Agent	10000	7	1	24	1.60	36.4/sec
TOTAL	30000	77	1	344	101.36	109.4/sec

Table 5.3: Retrieve Records and Agents Load Test

Retrieve Records and Agents after Authentication

Considering Table 5.4 retrieving data after authentication proved to be marginally slower than requests which do not need authentication. However, getting records was not nearly as slow compared to the first load test. This could be explained by the smaller magnitude of records that need to be retrieved. Those endpoints proved to be very fast and completely usable in a production environment. During the load test, the REST API reached full utilization again but the Postgres DB stayed comfortably at around 70% utilization. An increase in utilization for other components was not observed.

Request	Total Samples	Average (ms)	Min	Max	Std. Dev	Throughput
Get Record	10000	32	4	66	4.70	188.3
Get My Records	10000	20	3	53	3.22	188.3
TOTAL	20000	26	3	66	7.22	376.5

Table 5.4: Retrieve Records and Agents after Authentication Load Test

Log In

Regarding Table 5.5, logging in proved to be a relatively slow operation at an average response time of around 2 seconds, although the operation only reaches the REST API and the database. Another curious observation is that only the utilization in the REST API peaked to the maximum but the Postgres DB stayed with small to no utilization at all during the whole load test as well as the other components. This could be due to the secure generation of the signature in the JSON Web Token given to the front end. The load test was stopped after a convergence of results after 300 requests because the throughput was very low and the response times high.

Request	Total Samples	Average (ms)	Min	Max	Std. Dev	Throughput
Authentication	300	1986	101	2417	317.22	5.0/sec
TOTAL	300	1986	101	2417	317.22	5.0/sec

Table 5.5: Log In Load Test

Register Agent

Judging from Table 5.6 registering agents is an operation that is propagated through all services. For that very reason, we see longer response times averaging at about 2.5 seconds and increased utilization in many services. The REST API once again had a maximum utilization, the Transaction Processor was utilized at 10%, while the validator was utilized at 30%. The utilization on the Postgres DB was negligible as well as the rest of the components. We can conclude from the utilization that there is more computationally intensive domain logic that caused a delay in processing. The load test was stopped after 300 requests because the throughput was very low and the response times high.

Request	Total Samples	Average (ms)	Min	Max	Std. Dev	Throughput
Create Agent	300	2494	479	3692	532.92	4.0/sec
TOTAL	300	2494	479	3692	532.92	4.0/sec

Table 5.6: Register Agent Load Test

Create Record

Regarding Table 5.7 creating records was considerably faster than creating agents. The response times averaged about a second which is reasonable given the circumstances. Surprisingly, the REST API peaked at a mere 25% utilization whereas the Transaction Processor and Validator were more heavily utilized at 30% and 80%, respectively. The rest of the components did not exhibit an increase in utilization. The load test was stopped once again after 300 samples because the throughput was low and the test results have already converged.

Request	Total Samples	Average (ms)	Min	Max	Std. Dev	Throughput
Create Record	300	1149	43	1829	296.2	8.6/sec
TOTAL	300	1149	43	1829	296.02	8.6/sec

Table 5.7: Create Record Load Test

Update Record

Considering Table 5.8 updating records proved to be very fast given they reach all components of the system and perform read operations from the blockchain as well as adding blocks to it. The average response time for adding locations, linking and transferring records was around half a second with a minimum response time as low as 40ms. A curious detail in the load test is the larger than usual standard deviation which is even around the same as the average response time. This load test did not put the REST API under stress at all judging from the 12% utilization. This might be explained by the fact that the REST API does not modify the data in any way but just redirects it to the validator. The validator is also only slightly under stress at 20% utilization. An increase in utilization in the other components was not observed.

Request	Total Samples	Average (ms)	Min	Max	Std. Dev	Throughput
Add Record Location	338	486	41	1202	494.77	6.5/sec
Link Record	335	525	43	1225	495.421	6.5/sec
Transfer Record	327	541	62	1260	502.242	6.5/sec
TOTAL	1000	517	41	1260	497.998	6.5/sec

Table 5.8: Update Record Load Test

5.2 Testing

The project has been developed incrementally with different services at different times. For that reason, it was valuable during development to know if a service was not behaving as expected. There are different mechanisms set in place to indicate there is a problem with a service during development. If there was a problem with a service which caused it to fail to start, then all dependent services won't start as well. This is easily noticed and logs can be checked to see the cause of the issue. Another way of checking if everything works together is by health checks. Every service has a health check which indicates if the state of the service is not in a bad condition. Docker constantly checks that and notifies the user if the health check fails. Examples of failed health checks include the REST API or Event Subscriber not being able to connect to the reporting database.

Another way of testing if the components work is by sending a carefully chosen collection of requests to the REST API in a given order and checking if all of them pass. This was achieved by having a collection of requests which were sent in a given order using Postman[1]. This type of testing is a type of manual testing with little to no automation.

Finally, a small suite of e2e tests is developed to call different endpoints of the REST API and assert whether the response should be successful or failed with a particular error response code. This type of testing led to finding a few bugs when significant changes were made to the

[1]Postman is an API platform for building and using APIs.

system. Fortunately, the project was developed using Git and the GitFlow branching principle which argues that every new feature should be made in a different branch and merged back into master when done. Leveraging this strategy, I was able to track the bugs by comparing the feature branches with the master branch. The choice of manual and end-to-end[2] tests only one particular component was not arbitrary. It was a result of the limited time for the project. A more rigorous testing suite will include unit, integration and end-to-end testing on every service. The REST API was the target of tests because it interacts with many components and thus could identify problems with variety of services. What is more, testing a REST API was the most time-effective solution which brought the most value to the project.

5.3 Chapter Summary

This chapter provides insight into the process of evaluation of the speed of the system. It also describes the various ways of testing the application during development and the limitation of the testing suite.

[2]treating the back-end as a black box

Chapter 6

Conclusion

6.1 Challenges and Future Work

6.1.1 Data Model and Competitiveness

The design of the system had to take into consideration if the platform is deployed globally instead of focusing only on a few trusting companies. But how businesses will stay competitive using a supply-chain system where the data is essentially visible to other competitors in the same market? A natural solution might be to remove the identity revealing data from the blockchain and put it in a database. Then what happens if other competitors are able to deduce who stays behind a given digital identity in the blockchain by looking at the transactions they make? A solution might be moving even more data from the blockchain in favour of the traditional database. However, using this approach how does the blockchain contribute to the system in any meaningful way? There is always a trade-off between the type of data that is put on the blockchain and keeping a competitive advantage. Consequently, the process of designing a data model with an optimal balance of costs and benefits was a challenge. In the early stages of the project, everything was put on the blockchain, including agent names, emails, etc. After figuring out this model is not compliant with the GDPR rules (see Chapter 3.3.1) and exposes way too much information, a decision was made to keep only the identifications in the blockchain and move everything else to a transitional database. However, this approach diminishes the advantages of a blockchain system. On the next iteration, I brought back the initial data model but removed the identity revealing fields such as name, email and company and left the rest of the properties that will benefit the supply chain analysis such as location, material origin, weight and metal contents. The question of whether a company can guess the identity of a competitive company is still open. Speaking with a representative of the company in the business of recycling metals - Roster Ltd., provided an insight into this. It was noted that the weight of the purchased metal scrap is usually optimized for the carrier (rail, road or water transport) in order to reduce costs and it is not particularly different from company to company. They explained that the contents of metals do not deviate that much for a particular metal product as well and it cannot be an indicator of which company owns or manufactures it. Material type and origin are also not a reliable indicator according to the eastern-European company. The feedback I got concludes that the only somewhat revealing factor will be the location because there are not that many recyclers in a given area.

Further work on that matter should be done to improve the transparency of locations without compromising competitiveness. The locations can be encrypted with a key generated for each

user and stored in the database, while the cipher can be written on the blockchain. However, this approach prevents users from seeing the locations of records that are not owned by them but are included in their product's history. Another solution might be generating an encryption key for a record and sharing it with users that own products made as a result of that product. That way agents will be able to decrypt the locations of products that are not owned by them but are included in their product's historical analysis. However, those solutions rely a lot on having a database which is managed by a third party which defies some of the principles of distributed ledger technology.

6.1.2 Corporate Espionage

Another consideration when deploying the system globally is how members will stay anonymous with time. Imagine that Company A transfers materials to company B. Both parties will know each other's public keys after the transaction is completed, given they inspect the blockchain, even if the front-end hides this detail from them. From that point on, all transactions connected with those entities will be easily tracked by the other party. This could be both a positive and a negative feature. On one hand, it enforces trust between the companies and provides a simple way for both companies to monitor each other's activity. This could lead to offering better products as a provider or validating the origin of materials as a buyer. On the other hand, it will lead to exposing trade information to the trusted companies some of which might have bad intentions.

Further work is needed to display data that can be obtained from the blockchain but not through the front end. Creating a page where a circle of trusted companies is visible and having the functionality to monitor their transactions easily will provide better transparency and will lead to more benefits to all participants in the network.

6.1.3 Analysis and its Limitation

Although a great effort was put into providing as much analytical content as possible within the time frame of the project, there is a need for further improvement in that regard. Currently, the map showing different paths of products does not display the direction which makes it obscure if the data is not interpreted with care. Another addition might be more plots with insights from that data instead of relying on analysts to make them based on the data.

The greatest challenge in providing better analysis is that there is no guarantee of its correctness. Even if a product is included in the supply chain of another product, we do not know to what extent was the input product used in the development of the output product. If companies provide detailed information about the weight and content of their materials and their manufacturing/recycling process, the platform could be extended and thus providing insightful analysis of all records. However, because of the heterogeneous nature of WEEE this is neither cost-effective nor manageable.

6.2 Achievements

6.2.1 Metal Product History Analysis

The analysis of metal products was successfully implemented. It offers users of the system to see every metal product that was connected in any way to the final metal product, along with

an analysis of the origin, type, metal content, locations and others. All of that information is conveniently displayed in 2 world maps with markers indicating locations and heat maps indicating the intensity of locations at geographical points. The path of all metal products is also displayed on the map indicating their journey until the point they were either recycled or converted.

6.2.2 Multifunctionality of the Project

The project can be deployed in a variety of use cases depending on the current needs of the organisations that are interested to take part in it. It could be deployed in a single company with many worksites that handle different parts of the recycling process. The system will mostly benefit those companies with its immutability properties. It can be also deployed as part of three or more companies which would like to cooperate together and create a closed-loop recycling system. They will benefit from the transparency and trust the blockchain can offer. The system can also be deployed globally allowing many companies to form a network of closed-loop recycling systems and opening new markets for growth.

6.2.3 No Single Point of Failure

The system demonstrates a spectacular redundancy. Even if one component of the system goes down, the system will keep operating with little to no disruptions. If a NoSQL database is used, instead of the relational one, distributed through all the nodes, and a REST API is deployed at every node, we would achieve high availability because the failure of one node will not affect the system in any way. The only component which cannot be distributed is the Event Subscriber. However, in the event that it fails, only the state of the reporting database will be behind the newest changes on the blockchain but the system will be completely usable. The moment it comes back online, it will sync the on-chain and off-chain states of the system.

6.3 Reflection

The project was started without any prior knowledge in the decentralized field. Many hours were spent researching how blockchain technology works, the advantages and disadvantages of different blockchain types, along with choosing a framework that allows the development of industry-level applications. Afterwards, the framework that I choose had to be studied in-depth in order to leverage it into building the application. Although it was very challenging, it was also exceptionally rewarding to learn about those cutting-edge technologies. Looking back, I see the project as a unique learning opportunity. I expanded my knowledge in the blockchain domain as well as learning myself how to develop and deploy an application backed by a blockchain network. I also learned about the modern challenges in the supply chain industry as well as the environmental benefits of recycling metals.

6.4 Conclusion

Building a supply chain management tool which compels organisations to be part of the circular economy has many advantages for both the organisations and the environment. Companies can benefit from a more cost-effective solution to their supply chain needs. By doing so, they also

protect the environment by conserving natural resources and minimising the ore needed to be mined. In addition, blockchain technology enforces trust and transparency between the parties which is a key aspect of successful collaborations.

Bibliography

[1] Yaga D, Mell P, Roby N, Scarfone K. Blockchain Technology Overview. CoRR. 2019;abs/1906.11078. Available from: http://arxiv.org/abs/1906.11078.

[2] Hyperledger Sawtooth Documentation V1.2; 2022. Available from: https://sawtooth.hyperledger.org/docs/1.2.

[3] Hyperledger Sawtooth Documentation V1.1; 2021. Available from: https://sawtooth.hyperledger.org/docs/1.1.

[4] Vue.js JWT Authentication Flow; 2021. Available from: https://bezkoder.com/wp-content/uploads/2019/10/vue-jwt-authentication-flow.png.

[5] Awasthi AK, Li J. An overview of the potential of eco-friendly hybrid strategy for metal recycling from WEEE. Resources, Conservation and Recycling. 2017;126:228-39. Available from: https://www.sciencedirect.com/science/article/pii/S0921344917301933.

[6] Le Hoang L, Yamasue E, Okumura H, Ishihara K. MEMRECS—A Sustainable View for Metal Recycling from Waste Printed Circuit Boards. Journal of Environmental Protection. 2013 01;04:803-10.

[7] Norgate TE, Rankin WJ. The Role of Metals in Sustainable Development. Available from: http://www.minerals.csiro.au/sd/CSIRO_Paper_LCA_Sust.pdf.

[8] Martchk KJ. The Importance of Recycling to the Environmental Profile of Metal Products; 2000.

[9] Mayyas A, Steward D, Mann M. The case for recycling: Overview and challenges in the material supply chain for automotive li-ion batteries. Sustainable Materials and Technologies. 2019;19:e00087. Available from: https://www.sciencedirect.com/science/article/pii/S2214993718302926.

[10] Eduljee GH, Harrison RM. Electronic Waste Management. Issues in Environmental Science and Technology. Royal Society of Chemistry; 2019. Available from: https://books.google.bg/books?id=pcuuDwAAQBAJ.

[11] Scheijgrond JW. Extending producer responsibility up and down the supply chain, challenges and limitation. Waste Manag Res. 2011 Sep;29(9):911-8.

[12] Magrini C, Nicolas J, Berg H, Bellini A, Paolini E, Vincenti N, et al. Using Internet of Things and Distributed Ledger Technology for Digital Circular Economy Enablement:

The Case of Electronic Equipment. Sustainability. 2021;13(9). Available from: https://www.mdpi.com/2071-1050/13/9/4982.

[13] Kouhizadeh M, Sarkis J. Blockchain Practices, Potentials, and Perspectives in Greening Supply Chains. Sustainability. 2018;10(10). Available from: https://www.mdpi.com/2071-1050/10/10/3652.

[14] Foundation EM. Artificial Intelligence and the Circular Economy—AI as a Tool to Accelerate the Transition. Ellen MacArthur Foundation Cowes, UK; 2019.

[15] Standard SH. Federal Information Processing Standards (FIPS) Publication FIPS-180-4. US Doc/NIST. 2002. Available from: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[16] Greenspan G. The Blockchain Immutability Myth; 2017. Available from: https://www.multichain.com/blog/2017/05/blockchain-immutability-myth/.

[17] Esmaeilian B, Sarkis J, Lewis K, Behdad S. Blockchain for the future of sustainable supply chain management in Industry 4.0. Resources, Conservation and Recycling. 2020;163:105064.

[18] Shojaei A, Ketabi R, Razkenari M, Hakim H, Wang J. Enabling a circular economy in the built environment sector through blockchain technology. Journal of Cleaner Production. 2021;294:126352.

[19] Cheng Y, Hao H, Tao S, Zhou Y. Traceability Management Strategy of the EV Power Battery Based on the Blockchain. Scientific Programming. 2021;2021.

[20] RecycleGo partners with DeepDive Technology Group on blockchain technology for plastics recycling; 2022. Available from: https://recyclego.com/recyclego-partners-with-deepdive-technology-group-on-blockchain-technology-for-plast

[21] MongoDB. Sharding; 2021. Available from: https://www.mongodb.com/docs/manual/sharding.

[22] Everything you need to know about the ”right to be forgotten”; 2022. Available from: https://gdpr.eu/right-to-be-forgotten.

[23] Brown DRL. SEC 2: Recommended Elliptic Curve Domain Parameters. Standards for Efficient Cryptography. Available from: http://www.secg.org/sec2-v2.pdf.

[24] Dworkin M, Barker E, Nechvatal J, Foti J, Bassham L, Roback E, et al.. Advanced Encryption Standard (AES). Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD; 2001.

[25] Jones M, Bradley J, Sakimura N. JSON Web Token (JWT). RFC Editor; 2015. RFC 7519. Available from: https://www.rfc-editor.org/info/rfc7519.

[26] VueJS. What is Vues; 2022. Available from: https://vuex.vuejs.org/.

Glossary

batch Group of related transactions. In Sawtooth, a batch is the atomic unit of state change for the blockchain. For a batch with multiple transactions, if one transaction fails, all transactions in that batch fail. (The application is responsible for handling failure appropriately.). 16, 17

block A set of transactions (state changes for the blockchain) that is cryptographically linked to the previous block on the blockchain.. 17

Distributed ledger Another term for blockchain. A distributed ledger records transactions, in chronological order, that are shared by all participants in a network. Each block on the ledger is linked by a cryptographic hash to the previous block. 9

DLT Distributed Ledger Technologies. 9

EEE Electrical and Electronic Equipment. 6

Genesis block First block on the blockchain. The genesis block contains configuration settings and initial values for the network.. 14

HTTP/JSON HTTP/tcp connection protocol and using JSON data format. 17, 35

IoT Internet of Things. 9, 19

PBFT Practical Byzantine Fault Tolerance is a consensus algorithm where participants elect a leader to validate transactions. If malicious participants are less than 50% of the network, they are overruled by the other participants.. 16, 25, 26

Permissioned Blockchain Permissioned blockchains defy the original conception of the Bitcoin blockchain where everyone can read and write to the blockchain, and the ledger is transparent/public.. 6

PoET Sawtooth consensus algorithm that uses Proof of Elapsed Time (PoET) to agree on which participant should commit a block of transactions to the blockchain. PoET is a Nakamoto-style consensus algorithm that is designed to support large networks. It relies on secure instruction execution to achieve Nakamoto-style scaling benefits without the power consumption drawbacks of Proof of Work (PoW) consensus.. 16, 25, 26

PoS Proof of stake blockchain systems use the amount of stake a user has as a determining factor for new block creation. 13

PoW A consensus model where a mining node obtains the right to publish the next block by expending time, energy, and computational cycles to solve a hard-to-solve, but easy-to-verify problem (e.g., finding the nonce which, when combined with the data to be added to the block, will result in a specific output pattern).. 12, 13

REST API A component/service that adapts communication between a client and the validator using HTTP/tcp connection and using JSON data format. 17, 21, 34–36, 38

SQL Structured Query Language used for managing relational databases.. 35

state delta Event data that contains information about changes to the blockchain state.. 31

transaction A single change in the state of the blockchain.. 17

WEEE Waste from Electrical and Electronic Equipment. 6, 46

ZeroMQ ZeroMQ is an asynchronous messaging library, aimed at use in distributed or concurrent applications.. 17, 35

Appendix A

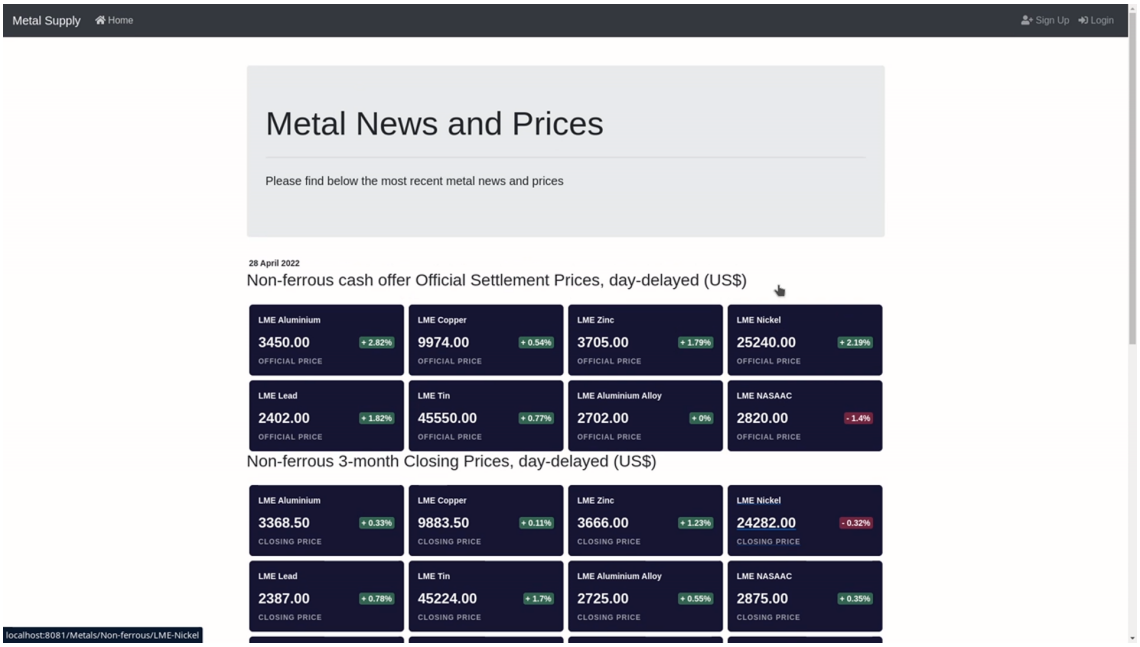

Figure A.1: Home Page

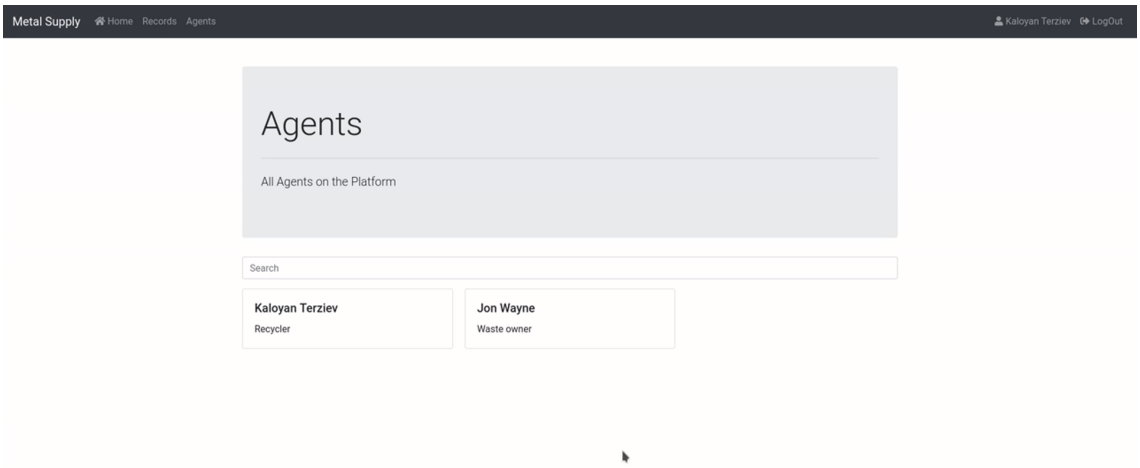

Figure A.2: Agents Page

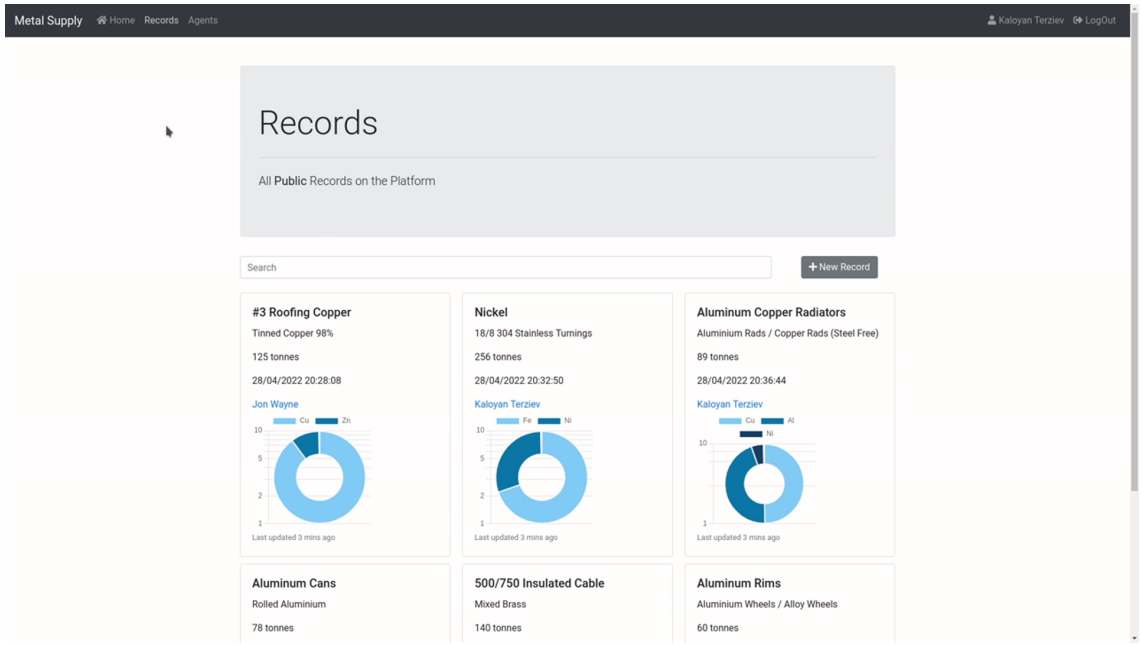

Figure A.3: Records Page

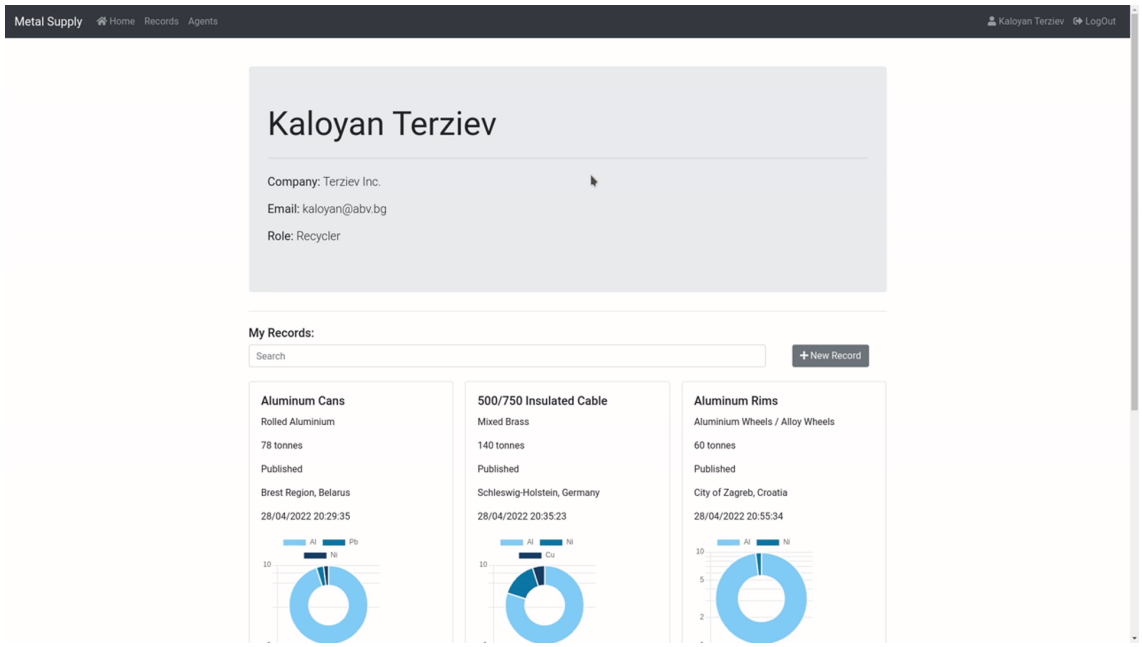

Figure A.4: Profile Page

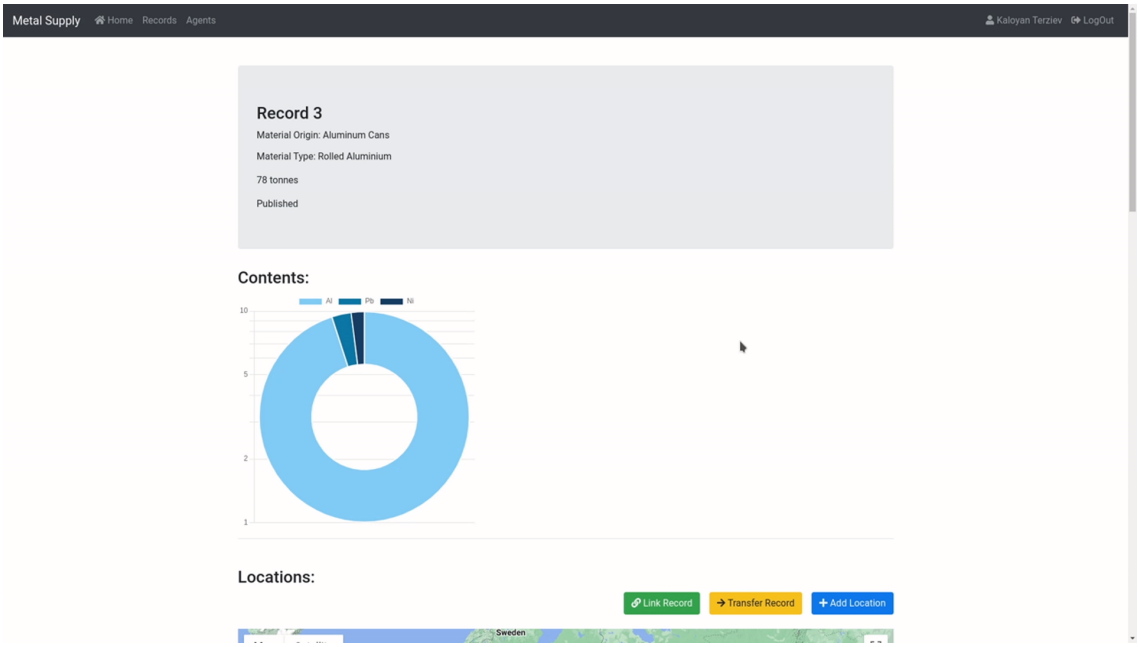

Figure A.5: Record Page

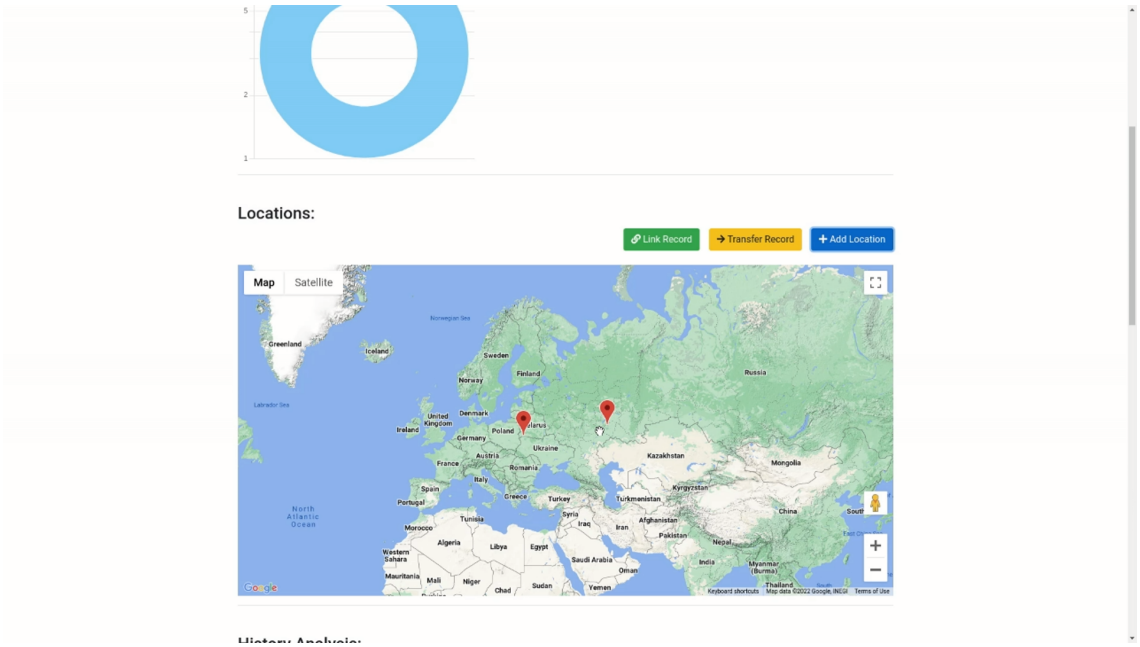

Figure A.6: Record Locations

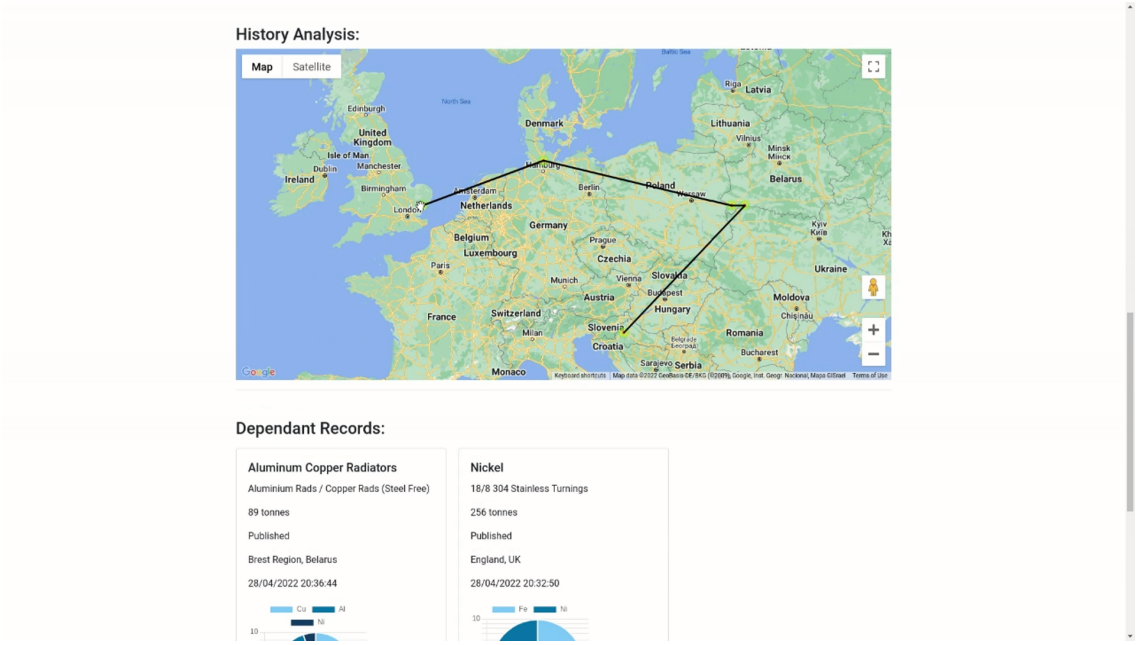

Figure A.7: Record History Analysis

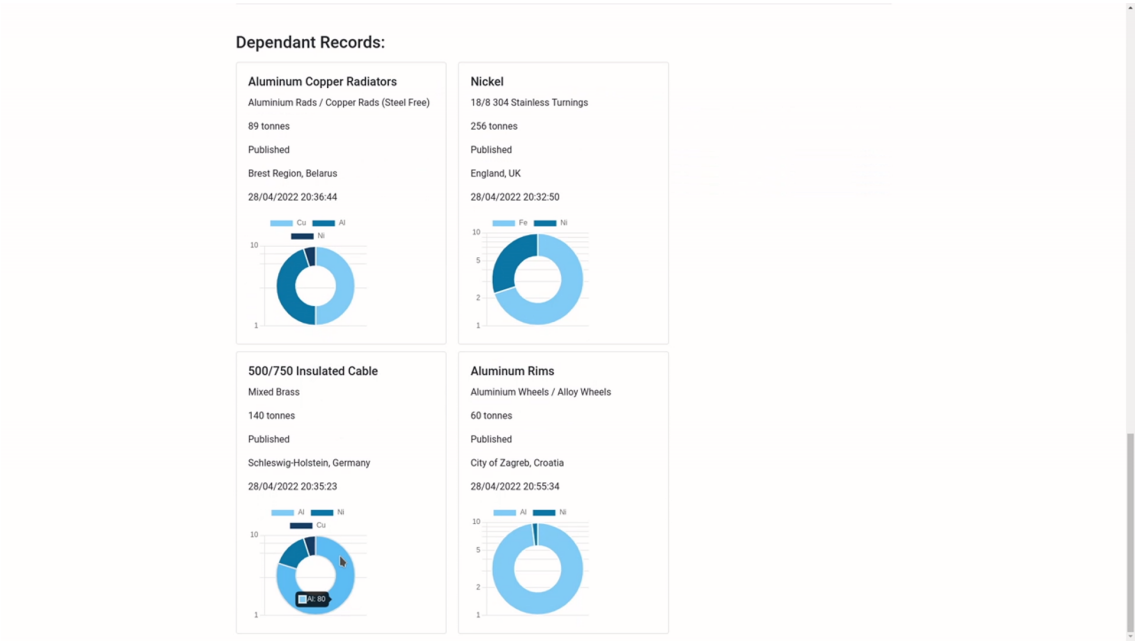

Figure A.8: Dependant Records