POLITECNICO DI TORINO

MASTER's Degree in INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

MASTER's Degree Thesis

eREG: Tokenization of Financial Instruments on a Public Blockchain

Supervisors
Prof.ssa Valentina GATTESCHI
Dr. Enrico TALIN

Candidate
Serigne Cheikh Tidiane Sy FALL

APRIL 2025

eREG: Tokenization of Financial Instruments on a Public Blockchain

Serigne Cheikh Tidiane Sy Fall

Abstract

This thesis aims to explore the process of tokenizing financial instruments, an innovation made possible by the recent Fintech decree and blockchain technologies. The research will focus on the preliminary analysis of the regulatory framework and the architectural technology choices that led to the selection of a public blockchain for the issuance of security tokens, the design of a customized smart contract and the development of a digital platform dedicated to financial instrument registry administrators. The advantages and challenges of implementing a decentralized financial management system will be examined, highlighting the role of regulatory compliance and security in system design.

Table of Contents

List of Figures

Chapter 1

Introduction

The increasing digitalization of financial markets, along with the adoption of Distributed Ledger Technology (DLT), has created new forms of investment and asset management. The 2023 Fintech Decree together with EU Regulation 858/2022, has established and defined the regulatory framework for the issuance and management of digitized financial instruments, also known as security tokens.

This thesis aims to illustrate how these regulations have been applied in the development of the **eREG** platform, a system designed for the tokenization of financial instruments, supported by a secure and transparent blockchain infrastructure.

The objectives of this project include:

- the analysis of the integration of fintech regulations in the development process of a blockchain-based platform and its application.
- the selection of a public blockchain, with a particular focus on security, transparency, and decentralization against the other types of blockchain.
- the study and development of a customized smart contract for the management of transactions and investor rights regarding financial instruments.
- the description of the creation of a dedicated platform for registry administrators, enabling the management and supervision of issued financial instruments.

1.1 Thesis structure

The goal of this document is to walk the reader through all the phases and provide an overall understanding of the work that has been done. In particular, an overview of the needs, the state of art and the requirements is supplied and also a comprehensive and exhaustive explanation of the implementation of the project.

Furthermore, to achieve what we just stated, the document has been divided into the following chapters.

- **Context & Background**

  In this chapter a context will be given to the reader and the necessary background to be able to understand the upcoming chapters.

  In particular, a description of the regulatory framework will be given, explaining the Regulation EU 2022/858 then the Fintech Decree and their relation. Additionally, since the thesis is discussing the tokenization of financial instruments, an overview of the main types of blockchain will be provided and we will discuss also the main types of tokens, from the NFTs to the fungible tokens.

- **The technologies**

  This chapter instead, will describe in detail the employed technologies. Starting from the Commercio.network blockchain, an overview of its module composition and the imported modules, in particular the Wasmd module, will be provided. Specifically, the chapter will focus also on the smart contract life management, describing all 3 phases.

  Furthermore, we will dig into the structure of the backend infrastructure, explaining each component and their implication in this project.

- **CW858 Smart Contract & eREG implementation**

  Here, the actual implementation of the project is described. As for the technologies, also in this chapter, we describe the implementation of the CW858 smart contract developed in Rust and then the implementation of the Golang Backend.

  Specifically, an overview of what is a smart contract and its main entry point will be provided to the reader. As well as how to write a smart contract within this scope. Successively, the implemented smart contract will be explained. However, the attention will be on the messages, their structures and what they actually represent.

  At the this stage, the chapter will introduce and describe the architecture design of the Backend and the implemented APIs to support a user to interact with the smart contract. Additionally, the focus will be on the business logic of the software application.

- **Tests and Results**

  Finally, in this chapter, an overview of how the tests were conduced within the different environments will be provided for both the smart contract and the backend.

  Ultimately, the attended results will be described to the reader and some representation of the Frontend will be provided for better insight.

Chapter 2

Context & Background

2.1 Introduction

The rapid advancement of blockchain technology has initiated an evolutionary era in the financial sector, introducing new concepts that challenge traditional financial paradigms. Central to this evolution is the process of **tokenization**, which converts rights or assets into digital tokens on a blockchain, thereby enhancing their liquidity, transparency and accessibility.

This chapter provides an in-depth examination of the regulatory framework governing tokenization, in particular the **Regulation (EU) 2022/858** and Italy's **Fintech Decree No. 25 of March 17, 2023**. These legislative measures represent significant steps towards integrating distributed ledger technology (DLT) within the European and Italian financial systems, aiming to foster innovation while ensuring robust investor protection and market integrity.

Understanding the underlying technology is crucial for appreciating the implications of these regulatory developments. Therefore, this chapter discusses the **fundamentals of blockchain technology**, beginning with its definition and key properties. The core characteristics of blockchain: **decentralization**, **immutability** and **distribution** are explained in detail. Decentralization eliminates the need for central authorities, immutability ensures data integrity, and distribution enhances the security of the network.

The chapter also examines the **various types of blockchains** and their respective applications. Public blockchains, such as Bitcoin and Ethereum, are open and permissionless, while private blockchains restrict participation to specific users. On the other hand, hybrid blockchains integrate properties from both public and private blockchains; meanwhile, consortium blockchains operate under the governance of a group of companies.

Furthermore, this chapter discusses also the variety of tokens that blockchain technology supports and the **process of tokenization**, detailing how assets are transformed into digital tokens on a blockchain.

Through this exploration of regulatory frameworks, blockchain fundamentals, token classifications and the tokenization process, this chapter seeks to offer a comprehensive understanding of how blockchain technology is changing the financial landscape.

2.2 Regulatory Framework

Figure 2.1: Regulatory Framework

About the regulatory framework, the European Union introduced a key decree to address the integration of Distributed Ledger Technology (DLT) in financial markets. This decree aims to establish a pilot regime for the issuance, trading, and settlement of financial instruments across all the EU members in a harmonized and compliant manner. However, in Italy, upon the proposal of the Minister of Economy and Finance, the Fintech Decree was issued by the Italian Council of Ministers and later signed by the President of the Republic for its promulgation. This decree is part of the Government's policies aimed at promoting technological innovation and regulating the use of blockchain and DLT technologies in the financial sector, in line with the country's digitalization and competitiveness goals. Hereafter, we'll go for a deeper observation of these two key decrees that have defined, among other things, the skeleton of our project.

2.2.1 Regulation (EU) 2022/858

As previously introduced, Regulation (EU) 2022/858 was enacted by the European Union to establish a pilot regime for regulating financial market infrastructures

based on DLT. This regulation provides a controlled environment that enables financial institutions to experiment with blockchain-based issuance and settlement of financial instruments. The main goal is to ensure the safe and effective integration of blockchains and DLTs into financial markets while addressing potential risks and challenges.

2.2.1.1 Overall Scopes and Goals

In general, the main purposes of the Regulation (EU) 2022/858 are:

- **Easing innovation:** by encouraging the adoption of solutions based on DLT technology inside the financial sector by building a legal framework sandbox.
- **Ensuring market integrity and stability:** by preventing market abuse establishing appropriate measures and ensuring the stability of financial markets during the transition to systems based on DLT.
- **Enhancing harmonization among EU members:** by providing a unified set of rules across EU member to ensure consistency.
- **Protecting investors:** by establishing measures to safeguard the rights and interests of investors within the DLT ecosystem.
- **Regulatory flexibility:** by allowing national authorities, like CONSOB, to promote the experimentation by granting exemptions from certain EU financial regulations but providing adequate limitations in the same time.

2.2.1.2 Structure of the Regulation

However, the Regulation (EU) 2022/858 is structured into various key components, in particular:

- **Definitions and scope:** this component defines the key terms and systems, the scope of application and the concerned types of financial instruments.
- **Pilot regime requirements:** this component instead, provides the conditions under which the financial institutions must handle market infrastructures based on DLT including the requirements for obtaining authorization.
- **Exemptions and safeguards:** it outlines the specific exemptions that may be granted to participants from the existing EU financial regulations, while defining the safeguards required to mitigate associated risks.

- **Oversight and supervision**: specifies the obligation of national and EU-level regulatory authorities in monitoring the adherence to the regulation, ensuring compliance and addressing potential violations.
- **Reporting and evaluation:** it outlines the requirements for participants to submit reports on their activities and performance within the pilot regime. It also establishes a process for evaluating the pilot's success in achieving its goals, enabling the regulators to identify possible improvements and make decisions about the regime's future.

2.2.2 Fintech Decree-Law 2023

Following the European framework, the Italian Council of Ministers issued the Fintech Decree to provide a tailored regulatory framework for the integration of Distributed Ledger Technology (DLT) within the national financial sector. This decree aims to address the specific operational, legal and technological needs of the national market by establishing guidelines for the use of blockchain-based systems. Moreover, according to Italia Fintech:
"the decree seems to be aimed at addressing, among other things, the following objectives: ***i.*** *Adjusting financial regulation in order to allow the use of new technologies, in line with the principle of technology neutrality;* ***ii.*** *Making DLT market infrastructures interoperable with those of the traditional financial system;* ***iii.*** *Allowing SMEs to issue debt instruments directly on the blockchain." [1]*

Going more into details, the Fintech Decree consists of several key chapters (*Capitoli*), each addressing fundamental aspects such as the authorization procedures, the operational requirements, the investor protection measures, the transition strategy and finally the description of the roles of the supervisory authorities. These chapters together aim to create an exhaustive and transparent framework that makes easier the experimentation from companies, ensures market integrity and grows investor confidence.
The upcoming section will break down the structure of the decree, providing an overview of each chapter and its specific focus areas.

2.2.2.1 Structure

The decree is structured into IX chapters including a total of 35 articles. Each chapter is dedicated to a specific regulatory theme providing detailed provisions and guidelines tailored to address key aspects as outlined below.

- **Chapter I: Definitions and Scope of Application**
  This chapter provides the definitions of key terms used throughout the decree

and specifies the scope of its application, including the entities and the digital financial instruments covered.

Some key definitions are, for instance, the individuation of three different categories of DLT market infrastructures, specifically:

i. DLT Multilateral Trading Facilities ("DLT MTFs") [1]

ii. DLT Settlement Systems ("DLT SS") [2]

iii. DLT Trading and Settlement Systems ("DLT TSS") [3]

Two additional key definitions are referred to ***Digital Financial Instruments*** and ***Register Administrator***. The following categories of financial instruments fall under the definition of digital financial instruments [1]:

a) ***shares*** referred to in Book Five, Title V, Chapter V, Section V of the Italian Civil Code;

b) ***bonds*** referred to in Book Five, Title V, Chapter V, Section VII of the Italian Civil Code;

c) ***debt securities*** issued by limited liability companies pursuant to Article 2483 of the Italian Civil Code;

d) ***additional debt securities*** whose issuance is permitted under Italian law ((as well as debt securities governed by Italian law issued by issuers other than Italian issuers));

e) ***depositary receipts*** related to bonds and other debt securities of non-domiciled issuers issued by Italian issuers;

f) ***money market instruments*** governed by Italian law;

g) ***shares or units of Italian collective investment undertakings*** referred to in Article 1, paragraph 1, letter l), of the TUF (Italian Consolidated Law on Finance).

While these categories outline the financial instruments described in Article 2, the Register Administrator is defined as either the issuer or a third party designated by the issuer, which is listed in the register referred to in Article 19, paragraph 1 of this decree.

- **Chapter II: Common Provisions for the Issuance and Circulation of Digital Instruments**
  This chapter outlines the general rules for the issuance and transfer of digital financial instruments. Here are some key aspects and references to the main articles that are involved in this project:

[1] DLT MTF is a multilateral trading facility operated by an investment firm or by an authorized market operator

[2] DLT SS is a settlement system that establishes transactions including DLT financial instruments against payment or delivery of these financial instruments

[3] DLT TSS is a DLT MTF or DLT SS that merges services performed by a DLT MTF and a DLT SS

- ***Article 3: Issuance and Transfer of Digital Financial Instruments***
  This article outlines the general rules for issuing and transferring digital financial instruments. It specifies the requirements for the issuance process and the conditions under which these instruments can be transferred between parties.
- ***Article 4: Requirements of the Registers for Digital Circulation***
  This article details the requirements for registers that manage the circulation of digital financial instruments. It includes the necessary information that must be recorded in these registers and the standards for maintaining the records accurate and up-to-date.
- ***Article 5: Effects of Entry in the Register***
  This article explains the legal effects of entering a digital financial instrument into the register. It clarifies the rights and obligations that arise once an instrument is officially recorded.
- ***Article 9: Establishment of constraints***
  This article discusses the creation of constraints on digital financial instruments. It describes the conditions under which these constraints can be established and their legal implications.
- ***Article 12: Issuance Information in the Register***
  This article details the information that must be included in the register when issuing digital financial instruments. It ensures transparency and accuracy in the issuance process.

- **Chapter III: Digital Financial Instruments Not Registered with a DLT TSS or DLT SS**
  This chapter focuses on digital financial instruments not registered with a DLT trading and settlement system (DLT TSS) or DLT settlement system (DLT SS). Some provisions that it includes are:

  a) ***Article 18: Issuance of Digital Financial Instruments Not Registered with a DLT TSS or DLT SS***
  This article defines the procedures for issuing digital financial instruments that are not registered with a DLT TSS or DLT SS. It specifies the requirements and conditions that have be met to issue financial instruments.

  b) ***Article 19: List of Register Administrators for Digital Circulation***
  This article establishes a list of register administrators for maintaining the registers of digital financial instruments. It also describes the criteria for selecting and approving these administrators.

  c) ***Article 23: Obligations of a Register Administrator***
  This article sets for the register administrator a series of obligations relating, among other things, to the general conduct, to the need to

ensure security, to the operational continuity and to the recovery of the register and also the obligations to inform the public about the operational procedures of the register itself.

- **Chapter IV: Supervision of the Issuance and Circulation of Digital Financial Instruments**
  This chapter assigns supervisory responsibilities to CONSOB and the Bank of Italy, specifying their respective roles and the obligations and requirements under the decree. It also defines the specific powers granted to them over the register administrators. Specifically, *Article 28* grants to CONSOB the power to dictate the implementation of the provisions, detailing their content and scope.

- **Chapter V: Provisions Related to the Application of Regulation (EU) 2022/858**
  This chapter focuses on the provisions related to the application of Regulation (EU) 2022/858. It addresses the authorizations, powers and roles of CONSOB and the Bank of Italy in enforcing and managing the regulation.

- **Chapter VI: Sanctions**
  This chapter focuses on sanctions and enforcement mechanisms to ensure the observance of the regulation. It specifies that the fines for the violations range from a minimum of €5000 to a maximum of €5 million. The chapter also describes the roles of the supervisory authorities in investigating breaches and imposing sanctions.

- **Chapter VII: Amendments to the Consolidated Law on Finance[4] and Final Provisions**
  This chapter includes modifications to the Italian Consolidated Law on Finance to incorporate digital financial instruments issued using DLTs. It also provides transitional provisions, specifying temporary measures for enrolling register administrators and detailing the roles of CONSOB and the Bank of Italy. However, within three years, these authorities must produce a report on the market impact and on the regulatory effectiveness to the FinTech Committee and Parliament.

- **Chapter VIII: Simplified Procedures for FinTech Experiments**
  This chapter specifies in *Article 33* that activities carried out as part of Fintech experimentation, within the limits set by the admission measures, are not considered as habitual performance of reserved activities under the definition of investment services and activities. Consequently, they benefit from an exemption from the requirement to obtain authorization.

- **Chapter IX: Final Provisions**
  This chapter covers the financial and procedural aspects related to the decree.

[4] as known as Testo Unico della Finanza (TUF)

In particular *Article 34* regulates the potential revenues generated from the sanctions outlined in Article 30 and includes a financial invariance clause to ensure no additional load on public finances. Lastly, Article 35 specifies the date on which the decree takes effect.

In general, Regulation (EU) 2022/858 and the Fintech Decree of March 17, 2023 are closely correlated since the Fintech decree was issued to adapt the European regulation to the Italian framework. The table below outlines the key differences between the two regulations to provide a clearer view of the regulatory framework.

Aspect	Regulation EU 2022/858	FinTech Decree of March 17, 2023
Objective	Establish a pilot regime for market infrastructures based on DLT	Implement urgent provisions on the issuance and circulation of digital financial instruments
Scope	Covers DLT market infrastructures for financial instruments	Focuses on digital financial instruments and simplification of FinTech experimentation
Regulatory Framework	Provides a temporary common EU pilot regime	Aligns with EU regulations and introduces national rules for digital financial instruments
Supervision	Supervised by national authorities and the European Securities and Markets Authority (ESMA)	Supervised by CONSOB and the Bank of Italy
Reporting	Requires reporting on market phenomena and regulatory effectiveness	Requires reporting to the FinTech Committee and Parliament

Table 2.1: Comparison of Regulation EU 2022/858 and the FinTech Decree

2.3 Blockchain and Tokenization

Given that our project focuses on the tokenization of financial instruments, this section will provide a comprehensive overview of blockchain technology, highlighting its key properties and the various types of blockchains. Additionally, it will discuss the different categories of tokens, with particular attention to security tokens.

2.3.1 Blockchains

A blockchain is a decentralized digital ledger that securely records transactions across a network of computers called nodes. Each transaction is processed by the nodes and registered into a block. These blocks form a chain of data as an asset moves from place to place or ownership changes. The blocks embed the exact time and sequence of transactions, and they link securely together to prevent any block from being altered or a block being inserted between two existing blocks.[2]. Some key properties of a blockchain include that it's *decentralized*, *immutable* and *distributed*.

- **Decentralized**:
  the control, in a blockchain, is not centralized under a single entity. Instead, the network operates on a distributed group of computers, called nodes, each of them carrying a copy of the ledger. This removes the need for intermediaries, such as banks or governments, and allows transactions to occur directly between participants. Decentralization makes blockchain more resilient to failures and attacks, since there is no single point of control in the system. This approach enhances trust among participants as decisions are made collectively through the consensus network. [3]

- **Immutable**:
  blockchains are designed to be tamper-proof. Once a transaction is recorded and added to the chain, it cannot be changed or removed. This is achieved through cryptographic hashing. In a blockchain, each block contains its own hash and the hash of the previous block, forming a cryptographically linked chain. For instance, if the content of a block is altered, the hash of that block changes, invalidating the hash reference stored in the subsequent block. This chain behaviour affects the entire blockchain, compromising its integrity. To restore the chain, the attacker would need to recalculate the hashes for all the subsequent blocks and gain control over the majority of the network's computational power, that is computationally and practically unfeasible in well established networks. This process guarantees the immutability and reliability of the data stored in the blockchain.[4]

- **Distributed**:
  a blockchain operates as a shared ledger that is distributed across all participants in the network. Each node(computer), depending on the type, maintains a full copy of the blockchain and independently validates every new transaction. This guarantees that is not a single entity controlling the data but all participants have equal access to the same information and power. Even if some nodes fail or are compromised, the blockchain continues to operate smoothly. Hence, the distributed nature of blockchain not only ensures high availability but also reduces the risk of data manipulation or loss. [2]

Apart from these fundamental characteristics of blockchain technology, there are several different types of blockchains. They mainly differ in terms of permissions, structure, and governance. Specifically, the main categories of blockchains are: **Public blockchains**, **private blockchains**, **Federated** and **Hybrid blockchains**.

Public blockchains as known as permissionless

This type of blockchain allows anyone to participate because, as the category name suggests, it's public. A public blockchain is fully decentralized, meaning that the control over the network is not in the hands of a single entity. In addition, anyone with an internet connection can join, participate, and interact by reading, writing, or auditing data. As a result, all the transactions and the records are publicly accessible, allowing users to review blockchain activity at any time.

Public blockchains are also referred to as permissionless blockchains since they enable any user to generate a personal address and become a node within the network without requiring approval or any kind of permissions. Furthermore, all nodes have equal access rights, which allows them to fully participate in creating and validating blocks. To achieve this last operation, in the case of a consensus [5] based on Proof-of-Work (PoW), they have to solve complex computations to validate a transaction. There are also other main consensus mechanisms like Proof-of-Stake (PoS) that is based on the amount of tokens staked by the node and the Proof-of-Authority (PoA) that is based on authorizations and is usually adopted in private blockchains.

One of the key advantages of public blockchains is their high level of transparency and security. Since a large number of nodes validate transactions, it becomes significantly more difficult for hackers to manipulate the system. Additionally, once a transaction is confirmed and recorded on the blockchain, it cannot be altered or modified, ensuring data integrity and immutability. In the other hand, the main disadvantages are that the network being this large, processing a block becomes slower and as it grows the speed rate decreases. Another negative side of this type of blockchain is that being completely decentralized, transparent and having anonymous transactions, it's very challenging to track frauds which results in limited protection for non-expert users.

[5]"A consensus mechanism is the programming and process used in blockchain systems to achieve distributed agreement about the ledger's state or the state of a data set. Cryptocurrencies, blockchains, and distributed ledgers benefit from their use because consensus mechanisms replace much slower and sometimes inaccurate or untrustworthy human verifiers and auditors." [5]

Figure 2.2: Public blockchain

Private blockchains

In this category of blockchain, only authorized nodes can participate in the consensus mechanism. In fact, a private blockchain is not open to the public, as the name suggests, but it's close to a private network of companies or organisations with a central entity that regulates its functions. Different from a public blockchain, in this type of blockchain, the nodes don't have all the same level of permissions and level of participation and each one of them is attached to a verified identity. Furthermore, private blockchains are by definition a lot smaller than public blockchains because of their closure to a few verified participants resulting into a faster blockchain that can process copious transactions per second.

Key advantages of a private blockchain are the speed, as just mentioned, because of its size and also the scalability of the network. The companies or organisations that use private blockchains can easily define the size of the network by adding or removing nodes. Another advantage is that since the credentials of the nodes are stored on the chain and since the identities of the nodes are verified, it makes easy to track frauds and therefore permits businesses to leverage blockchain technology while maintaining the desired level of transparency and privacy.

However, there are some disadvantages to face in private blockchains. From the small size of the blockchain we gain speed but we pay in the measure of security since it's easier for an attacker to gain control over the network and manipulate the transactions, action that is unfeasible on public blockchains. Another pain point is

that since the number of nodes is limited, the disruption of one of them can expose the whole network. Additionally, private blockchains are built breaking one of the main feature of blockchain: decentralisation. In fact, in this type of network, there is a central entity regulating the activities, thus breaking a fundamental principle.

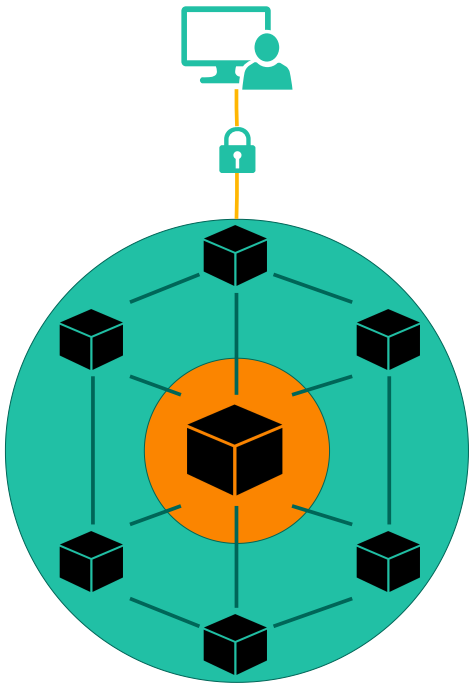

Figure 2.3: Private blockchain

Hybrid blockchains

The third type of blockchain is a combination between public blockchains and private blockchains, merging the best features from each side. In particular, hybrid blockchains are private networks within public blockchains, allowing companies to leverage both privacy and freedom simultaneously. In hybrid blockchains there is a central authority, like in private networks, that controls what data are meant to be kept private and what data can be shared in the public network. This central entity still cannot manipulate the transactions but only can define the level of privacy, hence the immutability property of a blockchain is still valid.

One key advantage of this type of blockchains is its structure, since it combines the best of public blockchains and private blockchains, being tempting for companies that want privacy on some transactions while having the possibility to work on a public blockchain. Usually this kind of blockchain suits highly regulated organisations like banks and companies that need to operate with a large public while keeping private some transactions, like real estate. In the other hand, a hybrid blockchain

can be very complex to maintain and to manage compared to the previous categories.

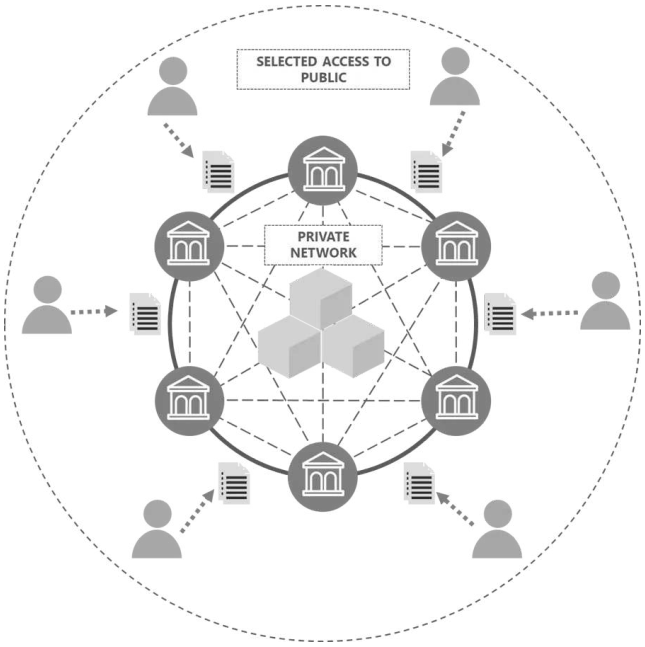

Figure 2.4: Hybrid blockchain [6]

Federated or Consortium blockchains

This last type of blockchain differentiates mostly from the private blockchain for the fact that there is not only a single entity that governs and controls the network but a set of companies. Indeed, consortium blockchains are built by a group of companies that reunite and operate on a blockchain to solve real world problems in the interest of the whole network. In a federated blockchain, the number of nodes is defined a priori and usually each company in the consortium controls a pre-defined subset of these nodes. In particular, these nodes participate in the governance of the network and have to vote for every block.

Moreover, the main advantages from this type of blockchain are that the blocks are verified by a consortium and at the same time hidden to the public, hence only the participants of the network can view them. Additionally, federated blockchains are scalable and fast, properties given by their contained size. Finally, having several companies that govern this network keeps it decentralized, maintaining the native property of a blockchain.

On the other hand, the drawbacks of this type of blockchain include limited transparency, as it is not accessible to the public, and governance challenges, as companies may face disagreements in decision-making.

In our project, we rely on **Commercio.network**, the company's blockchain.

Commercio.network was born as a federated blockchain open to 250 million of companies that can, each of them, manage a node. There are also 100+ companies that manage a validator node that have to sign every block to make it valid. [7]. Furthermore, Commercio.network is decentralized, has a governance of 100+ companies and private transactions between network's participants. During the years, ***Commercio.network became a public blockchain*** open to every subject with an internet connection.

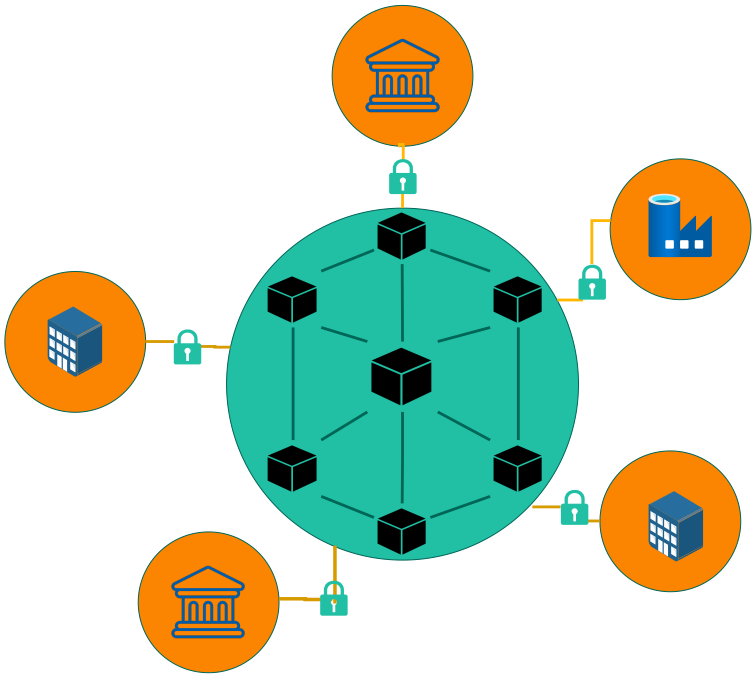

Figure 2.5: Federated blockchain

Finally, all four types of blockchains can be grouped into two main classes: **Permissionless** and **Permissioned**. Public blockchains belong to the first class while private and federated blockchains belong to the permissioned world. Hybrid blockchains belong to both classes since they combine both public and private blockchain properties, as we know.

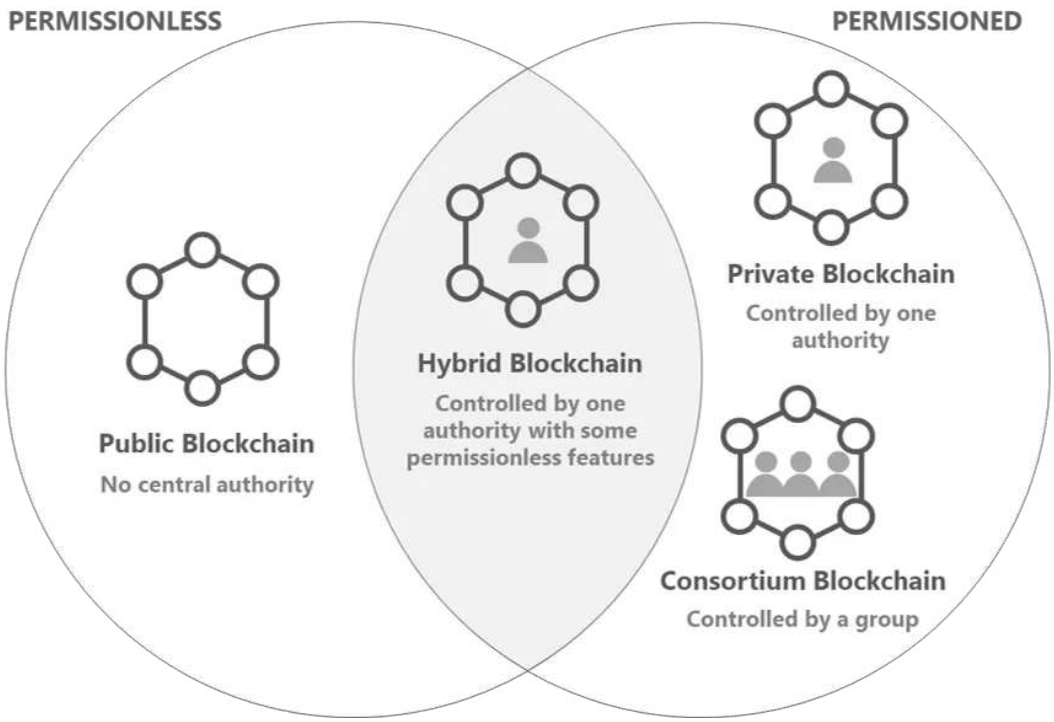

Figure 2.6: Permissionless and Permissioned blockchains [6]

In conclusion, here is a table to confront the main features from each type of blockchain and the main disadvantages, all together. It also includes some example blockchains for each type, to better visualize the general view.

Feature	Public Blockchain	Private Blockchain	Consortium Blockchain	Hybrid Blockchain
Permission	Permissionless	Permissioned	Permissioned	Permissioned and Permissionless
Access control	Open to everyone	Restricted to authorized participants	Restricted to consortium members	Combination of private and public
Transparency	All transactions are publicly viewable	Transactions are private	Limited transparency, depends on consortium rules	Can have both public and private data
Security	Highly secure due to complete decentralization	Secure due to limited access	Secure, but less decentralized than public blockchains	Security depends on design
Efficiency & Speed	Slower due to large network size	Faster due to limited participants	Faster than public blockchains	Speed and scalability depend on design
Governance	No central authority	Controlled by a single entity or authority	Controlled by a consortium of organizations	Can have centralized and decentralized elements
Drawbacks	High energy consumption, slower transactions	Less decentralized, potential security risks	Requires trust between consortium members	Complexity in maintenance and design
Examples	Bitcoin, Ethereum	Hyperledger Fabric, Quorum	R3 Corda	Komodo

Table 2.2: Key features of the 4 types of blockchain

2.3.2 Tokens and tokenization

As we know in blockchain, there are some objects called **Tokens**. A token is a digital representation of value, ownership, or access rights that can be traded, exchanged, or utilized within a specific blockchain ecosystem [8]. In general, we can distinguish two main categories of tokens: **Fungible tokens** and **Non fungible tokens (NFTs)**. The first category includes all kinds of tokens that are interchangeable and for instance, one Bitcoin token is perfectly equal to another Bitcoin token, they have the same value and utilization. Meanwhile, Non-Fungible tokens (NFTs) are unique tokens and cannot be replicated or exchanged for other tokens on a one-to-one basis. Each token is singular and different from the others. This fundamental characteristic makes them suitable for use in fields like Art, Collectibles and Real estate. One of the most famous use cases of NFTs is that of **CryptoKitties**. These NFTs are representations of unique digital cats on Ethereum's blockchain. Each Kitty has a different set of characteristics and a different price from the others. Users could buy them, collect them and breed them. In a short time, this game generated a fan base that spent millions of dollars on it.

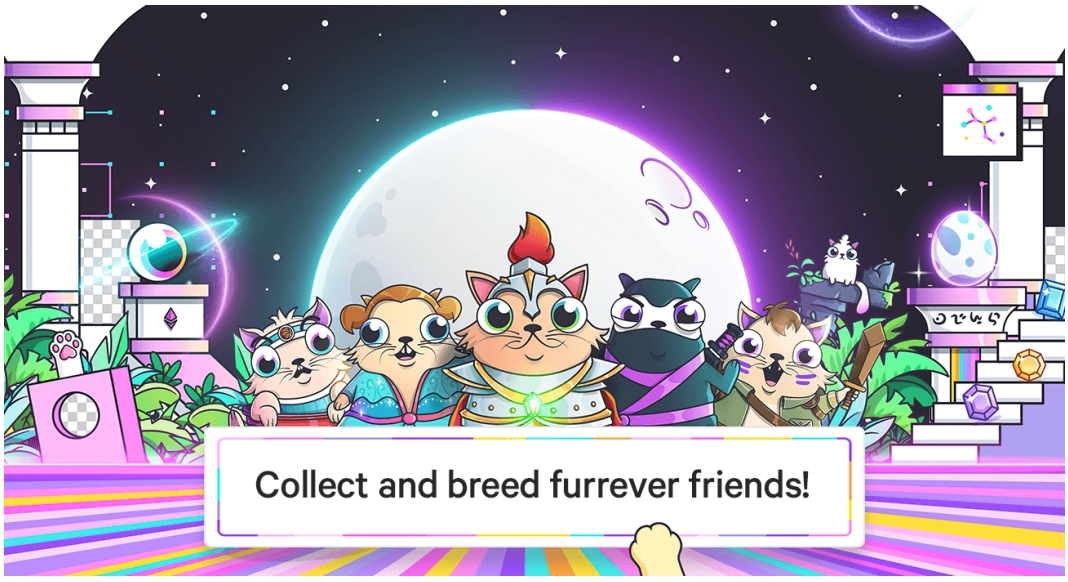

Figure 2.7: CryptoKitties [9]

Coming back to the Fungible tokens, we can extrapolate four main types of tokens: **Utility tokens**, **Payment Tokens**, **Stablecoins** and **Security Tokens**. However, it is important to notice that this types are not mutually exclusive, hence a token can belongs to more types since they can be used in different ways.

1. Utility Tokens

   These tokens are made to allow users to interact with specific applications and access specific products and services. They are not meant for trading

or used for investments. Their frequent use as a mean of transaction or as a mechanism to access premium features helps to demonstrate the general viability and sustainability of a blockchain platform. Usually, companies issue these tokens during Initial Coin Offerings (ICOs) [6] or Token Generation Events (TGEs) [7] as a way to raise capital, while simultaneously providing token holders with the ability to use or interact with the project's offerings. For instance, Commercio.network has a utility token called **Commercio Token (COM)** that is native. It has a limited capability of 60 millions tokens and this number can not be changed like BITCOIN [10].

2. Payment Tokens

   The second type of tokens are also referred to as cryptocurrencies of digital currencies since they are intended to facilitate the transfer of value within a decentralized network. They are used as a medium of exchange, enabling users to conduct transactions and settle payments without the need for intermediaries, such as banks or payment processors. Payment tokens are so useful because they provide quicker, safer and less expensive monetary transactions than more traditional means.

   The most famous and well-known payment token is **Bitcoin (BTC)**.

3. Stablecoins

   This token is designed to minimize price volatility and so meet the needs of some users. In fact, stablecoins are usually pegged to assets such as Fiat currencies, for example the US dollar, gold or a collection of currencies. These tokens aim to combine the stability of traditional assets with the benefits of blockchain technology, such as speed, security, and global accessibility.

   Some of the popular stablecoins available in the market are for instance, the **USD Coin (USDC)**, **Binance USD (BUSD)** and **Tether (USDT)**.

4. Security Tokens

   These tokens are investment tokens and represent ownership of real-world assets and they are subject to the same financial regulation. These tokens are the intersection of the blockchain technology with the traditional financial instruments, such as bonds, stocks and real estate. This means that security tokens offer to investors various financial rights, such as ownership, profit-sharing, dividends, or voting rights in a company. Security tokens are important because they carry many advantages for both investors and companies. One key

[6] ICO is a fundraising mechanism in blockchain where new projects sell their underlying utiliy tokens in exchange for capital.

[7] TGE is a process by which a blockchain project creates and distributes its native tokens. Similar to an Initial Coin Offering (ICO), TGEs are often used to raise funds, but the primary focus is on the technical process of token creation. TGEs involve the minting of new tokens on a blockchain platform, which are then allocated to investors, contributors, or other participants as outlined in the project's roadmap. This term emphasizes the technological aspect of token issuance rather than the fundraising angle.

advantage, is that security tokens remove the need of financial intermediaries as they allow companies to directly issue their products on the public market and investors to directly access the securities. Additionally, since there are less entities in the process, numerous fees that were paid to the intermediaries are no more due, the process of managing securities is faster and the complexity is lower. Moreover, security tokens allow companies to fractionate high-value assets giving the possibility of small investors to access the market, resulting into a bigger market. Smart contracts, make the exchange faster, automated and more safe for investors.

Since security tokens must comply with the regulatory framework of the assets they represent, until recently, no specific framework existed in Europe or Italy to regulate them. However, the introduction of Regulation (EU) 2022/858 and the Fintech Decree has provided European and Italian companies with the opportunity to experiment with this technology, as previously discussed. In this context, our project aims to tokenize financial instruments through the development of the **CW858 smart contract**, designed to issue securities in full compliance with the regulatory framework and we will discuss it in details later. Additionally, the associated platform will facilitate the interaction between issuers, register administrators and the smart contract on the blockchain ensuring seamless compliance and operational efficiency.

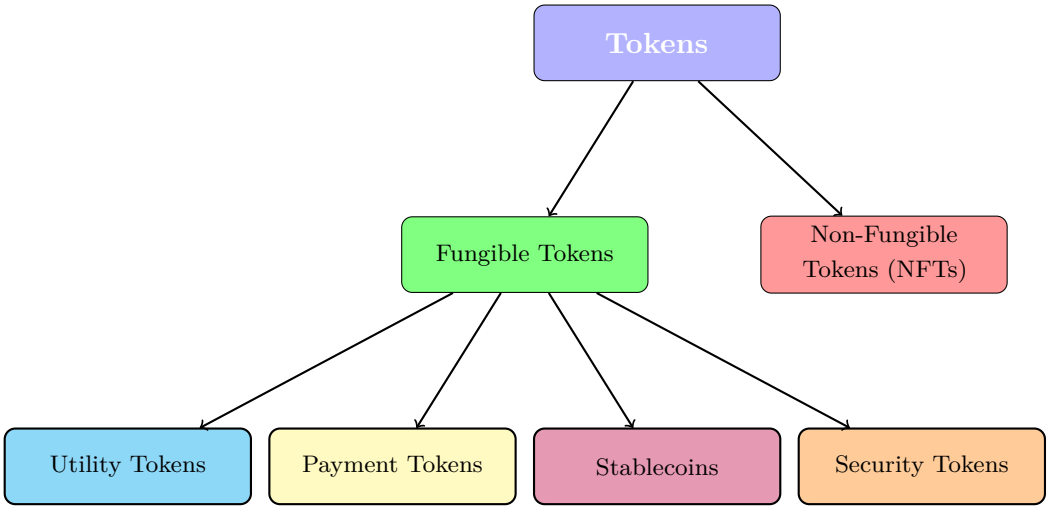

Figure 2.8: Classification of Tokens

In this context, we will briefly introduce the concept of **tokenization**. This process involves the conversion of rights or assets into digital tokens recorded and managed on a blockchain. This mechanism enhances the liquidity, transparency, and accessibility of various assets, facilitating more efficient and inclusive financial markets. Moreover, tokenization can be defined as: *the process of converting ownership rights or interests in an asset into a digital token on a blockchain. Each token represents a*

fraction or the entirety of the asset, allowing it to be traded and managed digitally. This process involves several technical and legal steps to ensure that the digital token represents the underlying asset and complies with the relevant regulations. These steps typically include:

- **Asset Identification:** Identifying the asset to be tokenized, such as real estate, commodities, securities, or intellectual property.
- **Smart Contract Creation:** Developing a smart contract that defines the terms and conditions of the token, including ownership rights, transfer rules and compliance requirements.
- **Token Issuance:** Minting the digital tokens based on the smart contract's specifications and recording them on the blockchain.
- **Distribution and Trading:** Distributing the tokens to investors or participants and enabling their trading on secondary markets or exchanges.

Furthermore, tokenization leverages the transparency property of blockchain technology and every transaction and change in ownership is recorded on an immutable block ensuring that all records are clear, traceable and secure. Additionally, this transparency not only reduces the risk of fraud but also enhances the trust among market participants. Moreover, tokenization with the use of smart contracts, that simplify operations and reduce dependency on third parties, enhances efficiency by automating processes.

With this general background established, the following chapters will delve deeper into the technologies utilized and the implementation of our smart contract, ensuring compliance with previously discussed the regulatory framework.

Chapter 3

The technologies

3.1 Commercio.network

As previously mentioned, our project relies on the company's blockchain, **Commercio.network**, also known as the "Documents Blockchain" and as the simplest way for companies to manage their documents leveraging blockchain technology.

In this section, I will briefly describe its structure, main components and its usage in this context.

Commercio.network is built upon **Cosmos SDK**, the world's most popular framework for building application-specific blockchains. This versatile framework offers a modular architecture that allows developers to create customized blockchain applications tailored to specific needs. It supports various programming languages and consensus algorithms, making it a flexible tool for blockchain development [11]. Key features of the framework include its modular design, the support for multiple programming languages and compatibility with various consensus mechanisms. For instance, Commercio.network is programmed in **Golang** and the actual version of the Cosmos SDK it's using relies on Comet BFT as consensus engine.

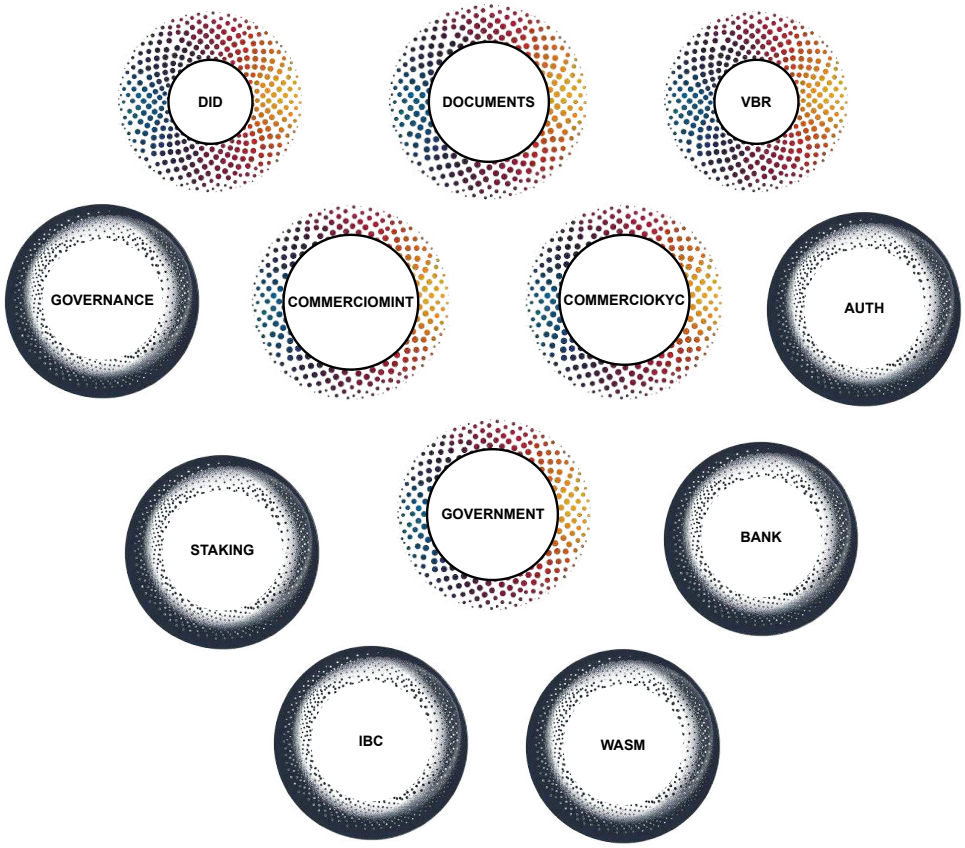

Figure 3.1: Blockchain's modules

Cosmos SDK's modular architecture enables us to compose different modules to build our custom blockchain applications. The main modules of the framework include:

- the **Auth Module**: for authentication
- the **Bank Module**: for token management
- the **Staking Module**: for the Proof-of-Stake mechanisms
- the **Governance Module**: for the community proposals and the voting mechanism

However, the foundation of blockchain applications lies in modules which encapsulate the business logic and in general, we can create and integrate these modules to define custom functionalities and composing them like building blocks to customize our business applications.

Specifically, at the core of the Cosmos SDK lies a robust infrastructure designed to support the key blockchain functionalities. This infrastructure includes a boilerplate

ABCI [1] implementation to interface with the consensus engine, a multistore for state persistence, a full-node server and query-handling interfaces. This core acts as the backbone of the SDK, wiring together various modules while maintaining efficiency and consistency.

Returning to our blockchain, Commercio.network is composed of a total of 6 custom modules besides the imported cosmos modules. These modules include:

- **Government module**: for handling the government address of the chain
- **Did module**: to allow the management of decentralized identities
- **Documents module**: to allow users to share documents to other users and receive receipts documents through the chain
- **CommercioMint module**: to allow the creation of Exchange Trade Position (ETPs) using COM tokens in order to obtain CCC tokens
- **CommercioKYC module**: a system to make sure to have a network of trusted participants, indeed KYC means "Know Your Customer"
- **Vbr module**: for allowing validators to get a recurrent reward each time a new block is proposed, even if such block does not contain any transaction.

It is straightforward to see that every single module serves a single purpose and business logic designed by the company. However, the modular architecture of the blockchain not only permits to include the Cosmos SDK's base modules or to implement custom modules but it also allows developers to **import modules** from other projects. This characteristic is very important, as it prevents us from reinventing the wheel. In this sense, Commercio.network, beyond others, imports two important modules: the **Inter-Blockchain Communication (IBC)** module and the **Wasm** module.

The **Inter-Blockchain Communication (IBC)** module is an interoperability protocol to allow different types of blockchain to communicate among themselves. In fact, IBC facilitates the transfer of assets, data and messages across independent blockchains, allowing them to interact without centralized intermediaries. This is essential for realizing the *Internet of Blockchains* where heterogeneous blockchains coexist and collaborate [12].

At its core, IBC operates through a structured architecture that includes these four main components: ***clients***, ***connections***, ***channels*** and ***packets***. The clients track the state of other blockchains, ensuring data accuracy. The connection component

[1] ABCI Application Blockchain Interface, is the interface between CometBFT (a state-machine replication engine) and the actual state machine being replicated (i.e., the Application). The API consists of a set of methods, each with a corresponding Request and Response message type.

manages the communication lifecycle, establishing and maintaining channels with other blockchains for data transfer. The channels act as paths for specific types of data or assets and lastly the packets carry the actual data or assets being exchanged. The protocol relies on cryptographic proofs to authenticate and verify transactions, ensuring security and integrity in cross-chain communication.

Moreover, the primary advantages of IBC include enhanced scalability and security across blockchain networks. It allows for cross-chain asset transfers and data sharing, fostering greater liquidity and utility. Additionally, the robust security measures ensure trustworthy interactions.

After this brief introduction to IBC, more focus must be committed to the **Wasm** module.

The Wasm module is an extension of the Cosmos SDK that enables the execution of WebAssembly (Wasm) smart contracts within Cosmos-based blockchains. It provides a robust framework for **deploying**, **executing** and **managing** Wasm-based contracts while ensuring security, flexibility and interoperability across the Cosmos ecosystem. One of the core components of this module is the **Wasm Runtime Environment**. This component provides the execution environment for Wasm smart contracts and it ensures that the smart contracts are executed securely and efficiently within the blockchain.

Furthermore, the wasm module integrates with the Cosmos SDK's Auth Module to handle authentication and authorization for smart contract interactions ensuring that only authorized users can execute the smart contract functions. For instance, this is useful when we want to limit who can deploy smart contracts on the chain, who can instantiate smart contracts and so on. The module has also integrated the Cosmos SDK's Bank Module to be able to manage token transfers and balances within smart contracts allowing them to handle financial transactions.

3.1.1 Smart contract lifecycle management

The key feature of the wasm module is the smart contract lifecycle management. In particular, this lifecycle management includes 3 main stages: deployment, execution and contract management.

- **Deployment:**

  Upon the development of a smart contract, a developer may need to deploy it on the blockchain to ensure it can fulfill its intended purpose.
  This module provides the necessary API that permits to deploy a smart contract

on blockchains that support it. There are few steps to follow for deploying a smart contract, that are:

1. compilation of the smart contract
2. generation of the ***.wasm*** binary
3. deployment of the contract using the wasm module's API

However, it is essential to generate a compact .wasm binary file for deployment, as the size of the binary directly impacts the GAS fees [2]. The larger the binary file is, the more GAS will be required to deploy it, leading to higher costs. Therefore, minimizing the file size is crucial to keep expenses as low as possible. For instance in our project, we have 2 ways to produce the optimized binary file.

The first way to produce the binary code is by running the following command:

```
RUSTFLAGS="-C link-arg=-s" cargo build \
--release --lib --target wasm32-unknown-unknown
```

In particular this command will build the Rust smart contract in release mode, targeting the WebAssembly platform (wasm32-unknown-unknown) and it will optimize the binary size by stripping unnecessary data (-C link-arg=-s).

The second method used for building a minimized binary code is by using a script on the *Cargo.toml* file that runs a docker container to optime the code.

```
[package.metadata.scripts]
optimize = """docker run --rm -v "$(pwd)":/code \
  --mount type=volume,source="$(basename "$(pwd)")_cache",target
    =/target \
  --mount type=volume,source=registry_cache,target=/usr/local/
    cargo/registry \
  cosmwasm/rust-optimizer:0.14.0
"""

# To run it
cargo run --optimize
```

This script runs a docker container using the cosmwasm/rust-optimizer:0.14.0 Docker image to optimize Rust code for WebAssembly. Moreover, this image

[2]GAS fees are transaction fees required to perform operations on blockchain networks and it represents the computational work needed to process and validate transactions or execute smart contract operations.

contains tools specifically designed to reduce the size of WebAssembly binaries, often by stripping debug symbols and performing other optimizations.

As previously mentioned, the integration of the Auth module in the Cosmos SDK enables the module to manage authorizations. Specifically, in this initial phase, it allows to define a set of addresses within the module's parameters that are authorized to deploy contracts on the blockchain. This mechanism empowers the blockchain governance to restrict the deployment of malicious contracts, offering an additional layer of protection for users.

However, this is not the only restriction that can be enforced. At this stage, it is also possible to define *execution constraints* for contracts, using the following flags:

- **instantiate-anyof-addresses** → Any of a set of addresses can instantiate a contract from the deployed code
- **instantiate-everybody** → Everybody can instantiate a contract from the deployed code
- **instantiate-nobody** → Nobody except the governance process can instantiate a contract from the deployed code

This stage of the deployment produces as output what is called **codeID**. A unique code that identifies on the blockchain the just uploaded smart contract.

- **Execution:**

At this stage, the smart contract has already been deployed on the target blockchain, such as Commercio.network. This state represents the core phase of its lifecycle management. Once deployed, interactions with the newly uploaded code can occur in three ways: **Instantiation**, actual **Execution** and **Querying** its internal state.

  - **Instantiation**

    After the deployment, the Instantiation of the contract is compulsory to be able to execute it's functions afterwards. Moreover, this phase is when we actually create an ***instance*** of the uploaded code on the blockchain. As result, the instantiation will produce a **unique address** identifying the contract's instance and we will use this address to interact with it.

    Essentially, this process requires 2 mandatory parameters: the ***code id*** previously produced to identify which contract to instantiate and a ***JSON-encoded message***.
    The second parameter is strictly related to the contract being instantiated,

in other words, the message must be formatted as defined inside the contract.

As a practical example of instantiation, consider the scenario where the CW20[3] smart contract has been deployed on our blockchain.
Assuming the code id is 8, the instantiation process on our blockchain would be as follows:

```
# set code_id var
CW20_CODE_ID=8

# set instantiation msg var
INIT_MSG='{"name":"CW20 token","symbol":"CWT","decimals":
    6,"initial_balances":[],"mint":{"minter":"did:com:1
    cjnpack2jqngdhj9cap23h4n3dmxcvqswgyrlc","cap":"250000"}}
    '

commercionetworkd tx wasm instantiate \
$CW20_CODE_ID "$INIT_MSG"  \
--label "Init Cw20 token"  --from $WALLET_CREATOR  \
--keyring-backend $KEYRING_BACKEND  \
--fees 10000ucommercio  --admin $WALLET_CREATOR \
--chain-id $CHAINID  -o json  -y
```

For convenience, we have defined two environment variables to store the two parameters. The *$INIT_MSG* variable represents the instantiation message specified during the contract's implementation, containing both mandatory and optional fields. Each smart contract defines a unique instantiation message within its code to meet its underlying objectives.

Also, within this context there are two flags used for the instantiation:

- ***label*** → to attach a human-readable name for this contract in lists
- ***admin*** → address or key name of an admin that will be able to exclusivly perform some *management* actions on the contract

– **Execution**

From this stage onward, users have full capability to interact with the smart contract through its instance. In particular, they can execute all publicly defined functions provided by the contract. However, these

[3]CW20 is the standard smart contract for fungible tokens within the CosmWasm ecosystem. Designed with various modifications, it derives its name and core functionality from Ethereum's ERC20 standard contract.

interactions remain subject to the internal permission mechanisms and authorization constraints embedded within the contract's logic.

Similar to the instantiation process, the execution phase also requires two mandatory parameters: the ***contract address***, which is generated upon instantiation and is used to target a specific contract instance and a ***JSON-encoded message***. As you might expect, this JSON-encoded message must adhere to the format defined within the contract's implementation to ensure proper execution. In particular, the mandatory fields, the types of the values and their structures must be respected.

As example, we might want to execute a CW20 token *mint*. The execution command on Commercio.network should look like:

```
# set contractAddress var
CW20_ADDR="did:com:1w27ekqvvtzfanfxnkw4jx2f8gdfeqwd3drkee3
e64xat6phwjg0sf3y3ja"

# set mint message
MINT='{"mint":{"amount": "250000", "recipient": "did:com:18
    h03de6awcjk4u9gaz8s5l0xxl8ulxjctzsytd"}}'

commercionetworkd tx wasm execute \
$CW20_ADDR "$MINT" \
--from $WALLET_CREATOR  \
--keyring-backend $KEYRING_BACKEND  \
--fees 10000ucommercio  \
--chain-id $CHAINID  -y
```

As previously mentioned, this message serves also as an example of authorization constraints embedded within the smart contract. Specifically, this message is used to issue new fungible tokens and only a predefined wallet address, marked as "*minter*" during the instantiation process, is authorized to invoke and execute this function. Every attempt of execution from not authorized wallets will result into an *unauthorized* error.

– **Querying**

Naturally, every smart contract maintains an internal storage system to preserve its state. The specific information stored depends on the contract's design and the data that may be relevant for users to retrieve at any time. To facilitate this, the wasm module provides the necessary endpoint to interrogate the internal state of a smart contract instance.

Also in this step, to query the internal state of a contract, the user needs to provide two mandatory data: the ***contract address*** of the instance to query and the JSON-encoded ***query data***.

In the case of the CW20 contract, a basic query would involve retrieving the stored information about the token. Specifically, some detail informations provided during the instantiation phase.

```
# set contractAddress var
CW20_ADDR="did:com:1w27ekqvvtzfanfxnkw4jx2f8gdfeqwd3drkee3
e64xat6phwjg0sf3y3ja"

# set the query data
TOKEN_INFO='{"token_info":{}}'

commercionetworkd query wasm contract-state smart \
$CW20_ADDR $TOKEN_INFO
```

This query will return the following informations:

```
data:
  decimals: 6
  name: CW20 token
  symbol: CWT
  total_supply: "250000"
```

- Management

  The WASM module not only provides endpoints for deploying and executing smart contracts, it also exposes functionalities for their management. In particular, it enables actions related to the contract usage and to the administration of the contract itself. Additionally, the module offers endpoints to query essential information about deployed smart contracts on the blockchain, such as their state and configuration, as well as to retrieve the module's parameters.

  For instance, the permitted management actions are as follow:

  - ***clear-contract-admin*** → Clears admin address for a contract to prevent further migrations
  - ***set-contract-admin*** → Sets a new admin address for a contract
  - ***grant*** → Grants particular authorizations to an address
  - ***update-instantiate-config*** → Updates instantiate configurations for a specific codeID

- ***migrate*** → Migrate a wasm contract to a new code version

On the other hand, the management informations that can be retrieved are more extensive. The available queries include:

- ***code*** → Downloads wasm bytecode for a given code id providing also a destination file.
- ***code-info*** → Returns metadata of a code id. Specifically, it returns the code id itself, the creator of the code, the hash and the instantiation permissions.
- ***contract*** → Returns metadata of a contract given its address. He re the returned informations are both the code id and the contract address, the creator address and the label specified during the instantiation. In addition, this endpoint returns also the block height where this contract instance was created and it's *ibc port id* if any.
- ***contract-history*** → Prints out the code history for a contract given its address. In particular, it returns all the performed operations on the contract and the executed message.
- ***list-code*** → List all the uploaded wasm bytecode on the chain
- ***list-contract-by-code*** → List the contract address of all the contracts instantiated from a specific code id
- ***list-contracts-by-creator*** → List the contract address of all the contracts instantiated by a certain wallet address.
- ***params*** → Returns the current wasm module parameters. In particular, the upload code and default instantiation permissions.

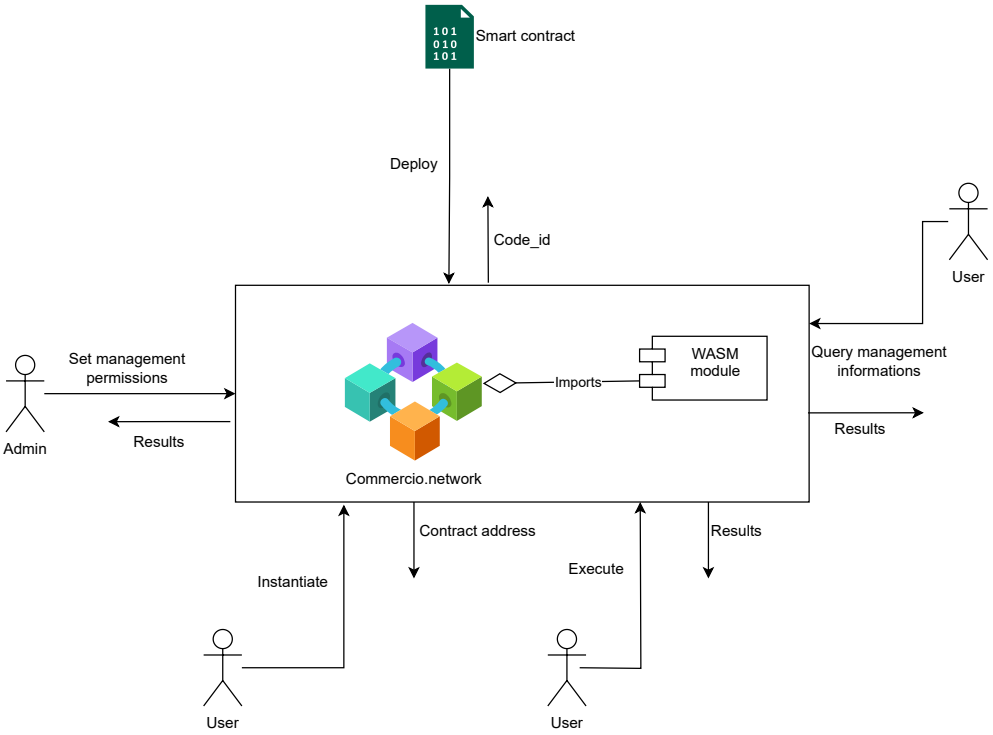

Figure 3.2: WASM module interactions

3.2 Platform Backend

Although our blockchain occupies a central role in this project, to provide users with an easy way to interact with it and the related smart contracts, a dedicated platform is required. To address this need, we introduce the **Hosted Wallet On-Premises Backend**. Specifically, this backend is developed in ***Golang*** and it is designed to facilitate direct interaction with our blockchain, Commercio.network.

However, the backend is not a standalone component; it is integrated inside a larger infrastructure that, among other things, handles user authentication and authorization, as well as the management of user wallets and private keys.

3.2.1 Structure

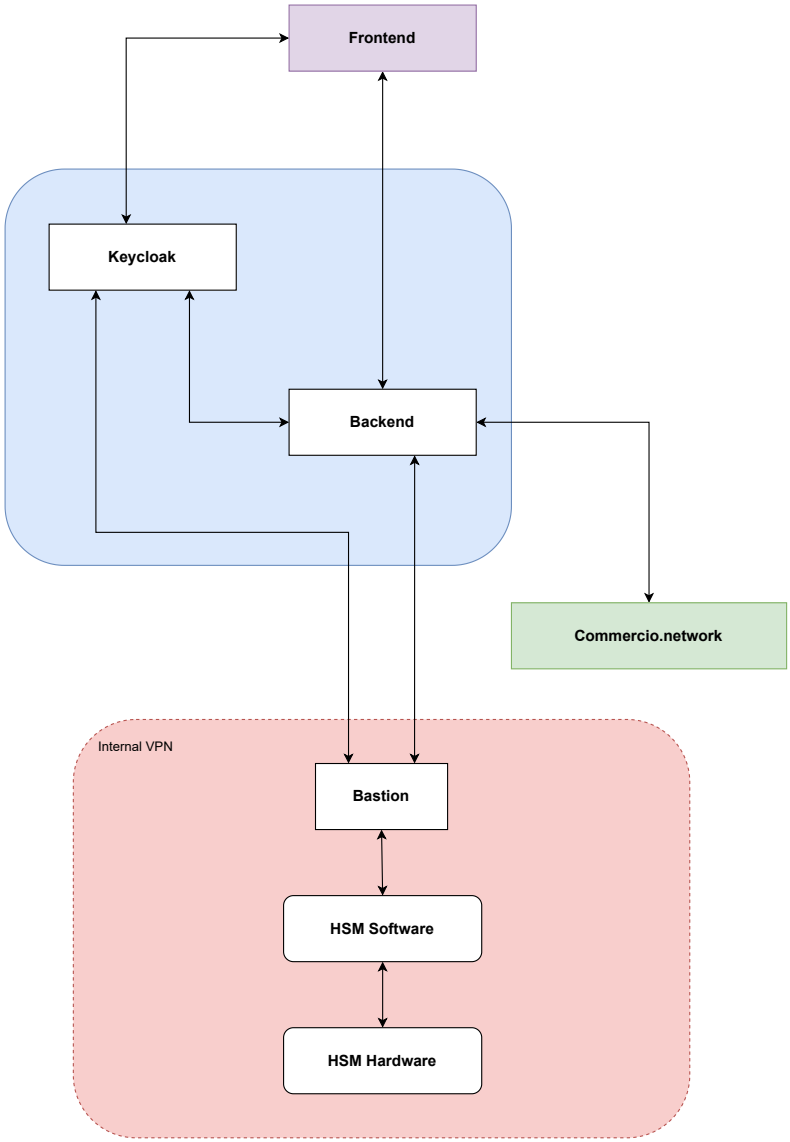

Figure 3.3: Backend infrastructure

As you can notice from figure 3.3 the infrastructure can be divided into several layers.

Starting from the bottom, there are three components: **Bastion**, **HSM Software** and **HSM Hardware** that are accessible only from inside the internal VPN as an additional protection for sensible data. Furthermore, any interaction from outside this layer must pass through the Bastion. This bastion host acts as a secure gateway, controlling access to the underlying network by being the only entry and exit point for the traffic. It functions as a checkpoint for external connections, reducing the risk of attacks and keeping the internal components protected from direct exposure.

This is essential for maintaining security, as it ensures that only authorized users can access the system. Also, the bastion permits to manage users key pairs via JWT tokens, the only authentication mechanism provided by the upper Keycloak.

Hardware Security Module (HSM)

Through the Bastion we access the replicated Hardware Security Module (HSM). This module ensures certified security for managing private keys and encryption in the Commercio.network blockchain. The private keys, representing user wallets, remain securely stored inside the HSM and they are never exposed, while signatures are generated and executed within the HSM. Interaction with these modules requires the **PKCS #11** specification and only public keys are exposed, which are also used to derive **Bech32 wallet addresses** via the HSM software.

Moreover, the HSM software service provides key management and signing functionalities via REST endpoints as follow:

- ***POST /raw_sign*** → Signs transaction messages using the user's private key and returns the signed bytes.
- ***POST /generate-keys*** → Creates a new wallet for a newly registered user.
- ***GET /user_details*** → Retrieves user wallet information using a JWT bearer token.

Frontend

As shown in Figure 3.3, the Frontend is the interface with which users interact directly. It allows them to issue security tokens, manage their balances and perform other actions depending on their permissions. Furthermore, the Frontend communicates only with the Backend, using the JWT bearer token provided by Keycloak to ensure secure access.

Backend services

As previously highlighted, **Keycloak** is our chosen open-source identity and access management provider, designed to handle both authentication and authorisation efficiently. By offering a centralized authentication endpoint, it simplifies the user identity management. Also, Keycloak supports widely used authentication protocols such as **OAuth 2.0** and **OpenID Connect (OIDC)**, enabling secure and flexible access control mechanisms[13].

OpenID Connect (OIDC) is built on top of the OAuth 2.0 framework as an identity layer. It allows client applications to verify user identities through an authorization server, standardizing the retrieval of authentication-related information. This facilitates Single Sign-On (SSO), allowing users to access multiple applications with a single authentication process, improving both security and user experience[14].

A key component of OIDC is **JSON Web Tokens (JWTs)**, which provide a secure and efficient method of transmitting authentication-related data. JWTs are digitally signed and contain structured claims about the authenticated user and session details. Each JWT consists of three parts:

- the **header**, which defines the token type and signing algorithm
- the **payload**, which includes user claims and session metadata
- the **signature**, which ensures the token's integrity and authenticity

The architecture enables secure, scalable and stateless authentication, establishing JWTs as an essential element in identity management and access control systems[15].

Furthermore, the following is an example of the sequence flow in which a user is redirected to Keycloak for authentication before gaining access to the protected API.

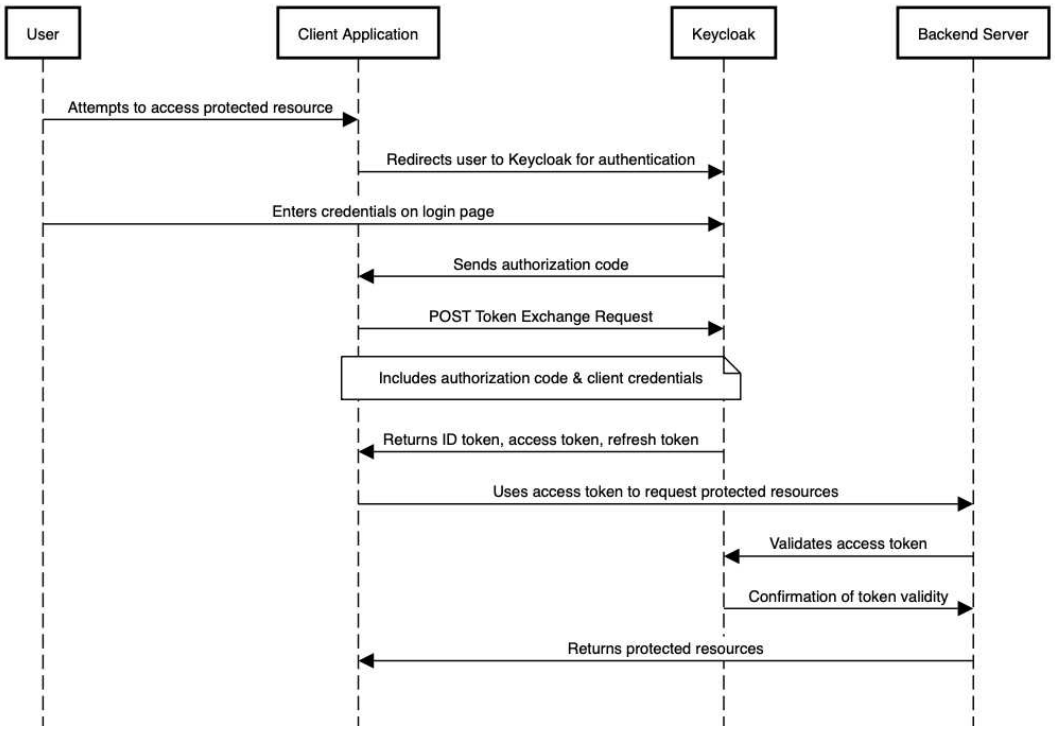

Figure 3.4: Sequence diagram

However, in the context of this project, the main component of our interest in the

entire infrastructure, a part of the blockchain, is the **Backend server** itself. This server acts as a middleware between the frontend and the blockchain.

In fact, the backend has been designed to expose protected endpoints to run transactions and execute smart contracts on the blockchain. Specifically, it interacts in two ways:

- **Frontend <–> Backend side**:

  During this phase of interaction, the backend exposes a set of API endpoints to the frontend, enabling users to perform various actions. These endpoints support the fundamental CRUD operations on the underlying resources, such as retrieving user information and issuing security tokens.

  Prior to processing any request, the backend first verifies the user's authentication. Once authentication is confirmed, it proceeds to handle and execute the corresponding logic. Additionally, the backend performs permission and authorization checks as required by the application logic before interfacing with the blockchain.

  In the following chapter, we will provide a detailed analysis of all endpoints and their underlying logic.

- **Backend <–> blockchain side**:

  In this context, the actions performed are straightforward and follow a sequential flow. For instance, when executing a transaction on the blockchain, the procedure should be structured as follows:

  1. build and prepare the transaction's message
  2. send the raw bytes transaction to the HSM to be signed
  3. HSM returns the signed transaction with the user's private key
  4. broadcast the transaction to the blockchain
  5. return the transaction response received from the blockchain to the client

  Unlike transactions, information retrieval requires fewer steps, as it does not involve signing with the HSM. Specifically, the process occurs as follows:

  1. prepare the query message
  2. execute the query on the blockchain
  3. parse the returned data from the blockchain
  4. return the queried informations to the client

Finally, to provide a comprehensive understanding of the infrastructure, the following global sequence flow illustrates the overall process, as depicted in Figure 3.3.

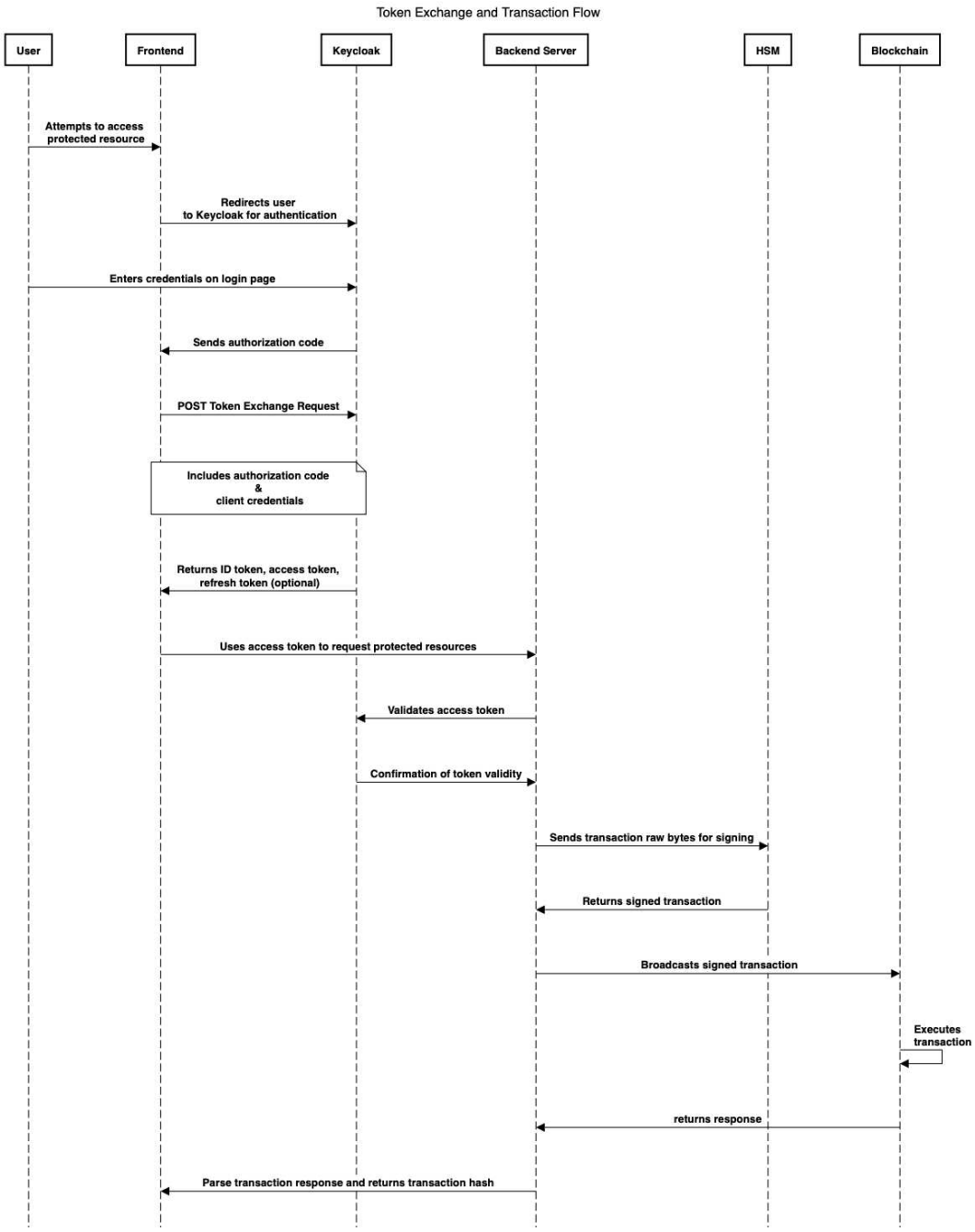

Figure 3.5: Infrastructure sequence flow

In particular:

1. **User Attempts to Access Protected Resource**
2. **Frontend Redirects to Keycloak** → The Frontend catches that the user is not authenticated and redirects the user to Keycloak for authentication.

3. **User Authenticates with Keycloak** → The user enters his credentials on Keycloak's login page. Keycloak validates the credentials and authenticates the user.
4. **Keycloak issues authorization code** → Upon successful authentication, Keycloak redirects the user back to the Frontend with an authorization code.
5. **Frontend Exchanges auth code for Tokens** → The Frontend makes a POST request to Keycloak's token endpoint to exchange the authorization code to receive an ID token, access token and optionally a refresh token.
6. **Keycloak Returns Tokens** → Keycloak responds with the tokens.
7. **Frontend Requests Protected Resource** → The Frontend uses the just received access token to request the protected resource from the backend server.
8. **Backend Server Validates Token with Keycloak** → The backend server validates the user access token with Keycloak to ensure its authenticity and validity.
9. **Keycloak confirms the token validity**
10. **Backend Server Signs Transaction with HSM** → Upon validation, the backend server sends transaction raw bytes to the HSM for tx signing.
11. **The HSM sign the transaction** → Using user's private key, the HSM sign the transaction and returns it to the backend server.
12. **Backend Server Broadcasts Transaction to Blockchain** → The backend server broadcasts the signed transaction to the blockchain.
13. **The blockchain executes the transaction and returns the response to the backend server**
14. **Backend Server Returns Data to Frontend** → The backend server parse the received transaction response and upon success returns tx hash. Otherwise it will return the tx error.

It is important to note also, that this sequence represents the use case of a non logged-in user attempting to access a protected endpoint that executes a transaction on the blockchain. Naturally, there are scenarios in which not all the entities involved in this process are required.

In conclusion, this chapter has provided a detailed overview of the fundamental technologies underlying this project, highlighting the main components and their roles.

In the next chapter, we will dive into the implementation of financial instrument tokenization, leveraging on our blockchain to enhance security and transparency. This

will involve deploying the **CW858 smart contract** and integrating our backend server to simplify interactions, ensuring smooth and efficient communication between users and the blockchain.

Chapter 4

CW858 Smart Contract & eREG implementation

Following to the background and technologies overview chapters, we are now able to proceed with the implementation of our CW858 smart contract. However, before delving into the technical details, a brief overview about smart contracts is required.

4.0.1 Smart contract overview

An American computer scientist who conceptualized a virtual currency called "Bit Gold" in 1998, Nick Szabo, first proposed the smart contract concept in 1994. He defined smart contracts as "***a computerized transaction protocol that executes the terms of a contract***" and that "***the general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries***" [16]. This concept didn't gain much popularity until 2015 after the introduction of the Ethereum blockchain which provided a platform for creating and executing smart contracts using the Solidity programming language.

Smart contracts permit to automatically execute agreements between parties without the need for external authorities or third parties. They use simple statements to ensure the validity of the pre-defined conditions and execute actions accordingly [17].

However, smart contracts can sometimes be challenging to understand, but a figurative example can help to provide clarity. A useful example can be the vending machine workflow: when a user inserts the correct amount of money and selects a product (input), the machine verifies that the right amount of money is provided

and the product availability before supplying the selected item (output). Essentially, under normal conditions, the same input will always generate the same output, regardless of which user initiates the transaction or any external conditions. This deterministic and automated execution can be seen also as the core functionality of a smart contract.

Once a smart contract is deployed on a blockchain, it becomes immutable and all the executed actions are tracked on the blockchain, enforcing transparency, security and agreement execution efficiency.

Furthermore, smart contracts permit us to establish agreements with other parties, formalize them into our preferred programming language and deploy them on the blockchain. This ensures that the contract cannot be altered due to its immutability property. In addition, they minimize the risk of human error in execution, thus guaranteeing an efficient execution [17].

In addition to the properties discussed in the previous chapter, the WASM module allows a contract to invoke other contracts synchronously. Since this framework adheres to the ***actor model***, messages are executed sequentially and deterministically. Consequently, if a contract X needs to call another contract Y, it must first complete its own computation, save its state and only then dispatch the message to invoke Contract Y. This structured execution flow effectively eliminates the risk of reentrancy attacks, a known vulnerability that haunts smart contracts written in Solidity [18]. Basically, this kind of attack takes advantage of the victim contract not persisting its internal state before calling another contract and suspending its execution.

Moreover, in WASM the top level message (meaning the initial message in the presence of sub-messages) is executed within an ***atomic transaction context*** for error handling. This means that if any sub-message returns an error, the top-level message also fails and returns an error triggering a rollback of all previously executed state changes.

For example, if the contract X first applies local modifications and successively invokes the contract Y which then performs additional changes, a failure in this last contract's execution will lead to the revert of the entire transaction.

Finally, all these considerations have guided us to our choice of relying on a smart contract written in **Rust** and managed through the WASM module for the tokenization of financial instruments.

4.0.2 How to write smart contracts

In Cosmwasm, in general, smart contracts have a specific structure. Fundamentally, they have 3 main entry points: ***Instantiate***, ***Execute*** and ***Query***.

The entry points are defined using an attribute macro that indicates to the underlying VM that the annotated function is an actual entry point. Specifically, the attribute macro is:

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn foo(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    msg: MyCustomMsg,
) -> Result<Response, ContractError> {...}
```

Actually, in this macro, in addition to defining `foo` as an entry point, the first part allows us to use the contract as a library later on [19].

- **Instantiate entry point**:

  This entry point is unique for every contract and it is called only once during the whole lifetime of a contract instance since every successful call produces a unique contract address that refers to an instance of the contract. For analogy, this entry point can be seen in OOP as a constructor function of our class, that in this case is the contract itself [19].

  The entry point definition is as follows:

  ```
  #[cfg_attr(not(feature = "library"), entry_point)]
  pub fn instantiate(
      deps: DepsMut,
      env: Env,
      info: MessageInfo,
      msg: InstantiateMsg,
  ) -> Result<Response, ContractError> {
      // instantiation logic
  }
  ```

  The entry point fundamentally has two types of parameters, the one predefined by ***cosmwasm-std*** and the one defined by the developer itself.

  Beginning with the predefined parameter type:

- **DepsMut**:

  This struct type provides us a *mutable* access to the **storage**, the Virtual Machine **API** and the **Querier**.

  In particular, the storage is a *trait* that provide access to the contract's persistent storage. Also the persistent storage can be accessed in read or write mode, depending on the parameters.

  The API instead, are callbacks to system functions implemented outside of the wasm modules and currently the trait just supports address validation and conversion.

  The Querier is a wrapper around *raw_query* that allows us to pass through binary queries from one level to another without knowing the custom format of the query.

- **Env**:

  This struct parameter supplies the function with the environment state information within which the contracting is executing.

  The provided informations are:

  * Blockinfo: this includes the block *height*, the absolute *time* of the block creation and the *chain_id*.
  * TransactionInfo: this information is set only when the contract execution is part of a transaction. The actual information it contains when it is set, is the index of the transaction inside the block.
  * ContractInfo: that should contains the contract address

- **MessageInfo**:

  This struct type provides some fundamental informations for authorizations and payments of an instantiation. In particular, it contains the *sender's* address of the transaction, in other words, the address who signed the transaction. It also contains the *funds* sent to the contract along with the transaction. The funds are an array of coins that the user might need to send to the contract depending on the instantiation's logic.

The last parameter is the actual parameter struct that the developer defines itself and according to the instantiation logic.

InstantiateMsg usually contains the default data of the contract and the initial values to store in the contract.

For instance, the *InstantiateMsg* struct of the CW20 contract we instantiated in the previous chapters includes the following:

- *"name"* → the name to associate to the CW20 token
- *"symbol"* → a specific symbol for listings
- *"decimals"* → accepted decimals for the token

- *"initial_balances"* → array of addresses and amount of token to send to them during the instantiation
- *"mint"* → optional struct that contains the token *minter* and optionally the token's total capability
- *"marketing"* → optional struct that contains marketing information such as project, description and logo.

- **Execute entry point**:

  This entry point differently from the previous one, it can be executed multiple times. Usually, in a smart contract, there are multiple execute messages for the entry point and there is a handler that based on the ***ExecuteMsg*** sent by the user, it triggers the correct function.

  Essentially, the function is similar to the instantiate entry point with the difference of the msg field.

```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    msg: ExecuteMsg,
) -> Result<Response, ContractError> {
    // match the msg
    // call the right execute function
}
```

  As mentioned before, contracts usually have multiple execution functions and one single entry point for execution. Consequently, to be able to handle all possible messages ***ExecuteMsg*** is an Enum of messages instead of a simple struct message. Each message in the Enum is a struct that defines its own fields for better handling its related function logic.

  For example, the CW20 contract has multiple execution messages and in the previous chapter, we chose the *mint* message as an example of a contract execution. In particular, this message has two fields:

  - *"amount"* → the amount of tokens to issue
  - *"recipient"* → the address to send the just minted tokens

  Supplying this struct message to the entry point, the contract will match it and call the related mint function to execute its logic.

- **Query entry point**: This entry point is for querying the internal state of a contract. Its function signature is as follows:

```
pub fn query(
    deps: Deps,
    _env: Env,
    msg: QueryMsg,
) -> StdResult<Binary> {
    // match the msg
    // call the right query function
}
```

Unlike the previous entry points, the query entry point is not provided with the *MessageInfo* struct parameter since there is not a transaction to sign or funds to send to the function. In addition, the entry point receives ***Deps*** instead of ***DepsMut*** since the storage can only be accessed *immutably*. In fact, here we just need to retrieve data from the internal storage of the contract and not to perform changes.

As for the Execute entry point, there are usually multiple query functions to retrieve different resources with only one entry point, thus ***QueryMsg*** is an Enum of query struct messages. Each struct defines the needed parameters to retrieve the desired resources.

For instance, in the previous chapter we used the *token_info* query message to retrieve default data stored during the instantiation process.

Essentially, defined these three main entry points, their messages, and their functions, we have a fully deployable smart contract.

4.0.3 CW858 implementation

This project focuses on the tokenization of financial instruments. Thus, for the tokenization process, we need to implement a smart contract that will represent our security token.

As mentioned multiple times, our work is within the Cosmos ecosystem, where no standardized contracts for security tokens currently exist. Specifically, we have CW20 for fungible tokens and CW721 for NFTs as the two main smart contract standards for tokens. Additionally, these two contracts were implemented following two main standard contracts from Ethereum, respectively ERC20 and ERC721.

In our context, we followed the same logic implementing our security token **CW858** from an Ethereum's standard contract, the **ERC1400**.

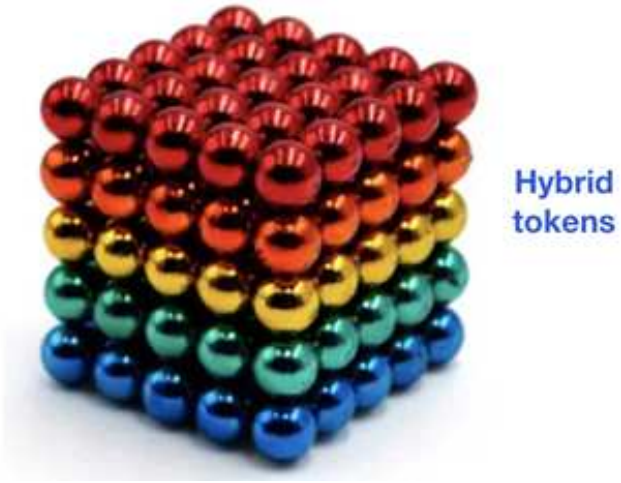

Figure 4.1: Hybrid Tokens [20]

Actually, the ERC1400 is a **Hybrid token**. A type of token that has not been mentioned until this point. Specifically, this type of token is a combination of fungible tokens and non-fungible tokens. Each token belongs to a specific ***partition*** and the tokens of the same partition are fungible; meanwhile, the tokens belonging to different partitions are distinguishable from each other.

Furthermore, for the implementation of our CW858 smart contract, we first implemented the basic security token following the ERC1400, applied the Fintech decree requirements and then implemented the constraints.

In the following sections, we will outline the implemented functions, their underlying logic and their essential role in achieving tokenization. Additionally, since the entry point signatures have already been described, the focus will be on the messages and the underlying logic.

Before analysing the messages in detail, it is important to introduce some roles. Following the Fintech requirements, we defined 3 major roles: **Register Administrator (R)**, **Issuer (E)** and **Token Owner (T)**.

Each role has different privileges and permissions from the others. In particular, the Register Administrator has full access and it is the contract **owner**, the issuer instead, has some limitations on the contract management and some administration features and it can be a **controller** and/or a **minter**. Meanwhile, the token owner has only the possibility to operate on its own tokens.

Instantiate message

As you already know from the previous section, this message is fundamental and requires the default data that are crucial for the contract to initialize its storage.

```rust
pub struct InstantiateMsg {
    pub name: String,
    pub symbol: String,
    pub decimals: u8,
    pub granularity: Uint128,
    pub initial_controllers: Vec<String>,
    pub default_partitions: Vec<[u8; 32]>,
    pub mint: Option<MinterResponse>,
    pub marketing: Option<InstantiateMarketingInfo>,
}
```

At first observation, you may notice some similarities with the instantiation message of the CW20 contract, this is due to the fact that the CW858 indeed implements all the messages that are implemented in the CW20 contract.

In particular, the message fields are as follows:

- *name* → the name of the security token
- *symbol* → its associated symbol
- *decimals* → the maximum accepted decimals for the token
- *granularity* → its maximum granularity. For example: token amount when doing a transfer must be a multiple of the granularity.
- *initial_controllers* → list of addresses to set as controllers during instantiation.
- *default_partitions* → list of default partitions of the token. It also can be left empty, but this will lead to some limitations later on.
- *mint* → optional struct that contains the token *minter* and optionally the token's total capability
- *marketing* → optional struct that contains marketing information such as project, description and logo.

Furthermore, this message permits us to set the basic data of a security token, set the initial controllers known as issuers and set the default partitions. The owner of the contract, the Register Administrator, is the address that instantiates the security token.

As the Register Administrator must be listed as in *Art. 19: List of Register Administrators for Digital Circula- tion* of the Fintech Decree and have specific characteristics, we permit the instantiation of a new security token only to one address that we set through the backend (we will see it later).

Additionally, in the related instantiation function, the contract's internal storage is initialized with the supplied values and the controllers are automatically added to a ***whitelist***.

Execute messages

As previously mentioned, the CW858 contract implements, besides others, all the CW20 messages. Hereafter, the actual main execute function in this context.

Mint

This execute message serves to issue new tokens and it is defined as follows:

```
pub enum Cw20Msg {
    ....
    Mint {
        recipient: String,
        amount: Uint128,
    },
    ....
}
```

Essentially, it includes two fields: the *recipient* which is the address that will receive the newly minted tokens and the *amount* that indicates the quantity of tokens to be minted.

Logically, the related function can be executed only by authorized users. In particular, only the minter (Issuer) and the contract owner (Register Administrator) can call and execute it. Any other sender [1] will encounter an unauthorized error.

Burn

The Burn message is the opposite of Mint. However, it removes the tokens from the wallet of the sender and also from the total supply, which indicates the amount of circulating tokens.

```
pub enum Cw20Msg {
    ....
    Burn {
        amount: Uint128,
    },
```

[1] Sender refers to the user that signed the transaction and so the address that is effectively executing the contract.

```
    ....
}
```

Differently from what happens in Ethereum, where the tokens are sent to an inaccessible address, here the tokens are simply erased from the sender's wallet.

Burn From

From the management point of view, the simple Burn message can represent a limitation. As an register administrator, I may need to burn tokens from a wallet as a consequence of an error during mint or other operations or even for legal purposes.

The message structure is as follow:

```
pub enum Cw20Msg {
    ....
    BurnFrom {
        owner: String,
        amount: Uint128,
    },
    ....
}
```

In fact, in addition to the amount field, there is also the *owner* field which indicates the address of the owner of the tokens to be burnt.

As you might expect, only authorized users, like the Register Administrator, can execute this function.

Transfer

This message, instead, is for transferring tokens from the sender's wallet to another wallet.

```
pub enum Cw20Msg {
    ....
    Transfer {
        recipient: String,
        amount: Uint128,
    },
    ....
}
```

Also in this case, in addition to the amount field, there is the recipient field which is used to indicate the wallet where to move the tokens. This command will succeed only if the sender has sufficient balance in his wallet.

Moreover, this execute message will not trigger any other actions even if the recipient address is another contract.

Transfer From

In the management of financial instruments, the authorities must have the possibility to execute forced actions in some cases. We have to provide these authorities, such as the Register Administrator, the capability to coercively move tokens from a specific wallet to another for various reasons.

```
pub enum Cw20Msg {
    ....
    TransferFrom {
        token_holder: String,
        recipient: String,
        amount: Uint128,
    },
    ....
}
```

In fact, the message has the same fields as the basic *Transfer* except for the additional *token_holder* field. This extra field effectively specifies the wallet from which to transfer the tokens.

Clearly, it can be executed only by the Register Administrator and the Issuer.

Update minter

As only one minter can be established at a time, this message enables the assignment of a new minter when necessary.

The conditions are that: there must be an actual minter and only this minter can execute this message.

```
pub enum Cw20Msg {
    ....
    UpdateMinter {
        new_minter: Option<String>,
    },
    ....
```

```
}
```

Additionally, setting a new minter is not mandatory, as indicated by the optional field. Therefore, removing the minter will permanently delete the contract's minter, preventing any further token issuance.

Furthermore, beyond this CW20 related main execute messages, there are the following messages that are specific to the security token.

Controllers management

Controllers are subjects that have more privileges than normal users. As mentioned before, issuers can also act as controllers. In particular, controllers can perform actions on other wallets different from their own wallets.

These special users are stored inside an array in the internal storage of the contract and the following messages allow us to manage that array.

```
pub enum ExecuteMsg {
    ....
    SetControllers{
        controllers: Vec<String>,
    },
    RevokeController{
        controller: String
    },
    ....
}
```

In particular, the *SetControllers* message is employed to set a new list of controllers. In fact, it completely replaces the previous list. It can also be left empty, meaning that only the Owner will have the privilege to operate within the other users' wallets.

While *SetControllers* sets a new list of controllers, *RevokeController* removes a controller from this list.

Both these messages can be executed only by the Owner, meaning that only the Register Administrator has the authority of upgrading one to controller or downgrading one from it.

Whitelist management

A key restriction measure employed in the security token is the *whitelist*. In fact, a user is able to move tokens from his wallet or to receive tokens only if he is

whitelisted. For example, the *mint* message to succeed, both the recipient address and the sender must be whitelisted. The same behaviour is implemented by all the other execute messages that need to perform some token movement.

Furthermore, the following messages are used to manage the whitelist.

```
pub enum ExecuteMsg {
    ....
    AddWhitelist{
        account: String
    },
    RemoveFromWhitelist{
        account: String
    },
    ....
}
```

AddWhitelist adds an address to the whitelist while *RemoveFromWhitelist* removes an address from it.

This message can be triggered only by the Register Administrator or by the Issuer.

However, the whitelist is connected with the **KYC (Know Your Customer)** module of the blockchain. Whenever there is an attempt to add an address to the whitelist, the contract calls the KYC module to validate the user's KYC membership.

This ensures that every token owner is well known and all the necessary information to identify the physical person connected to that address is valid and available.

Contract administration

Besides these execute messages, there are also some "utility" messages that serve for the token management.

- ***Renounce Control***

  This message in particular, when triggered all the privileges related to the controllers are lost permanently. No controller will be able to perform privileged functions thereafter.

- ***Renounce Issuance***

  This message instead, disables permanently the token issuance. Once executed, no more tokens can be minted neither from the Register Administrator neither

from the minter.

- ***Transfer Ownership***

  Since the Register Administrator is nominated according to specific rules from the Fintech Decree, there may be the need to change it from one address to another. This message serves this need, it permits to the actual Owner to set a new one and transfer all the privileges to this new Owner.

- ***Renounce Ownership***

  As for the previous renounce messages, this message also deletes permanently the ownership of the contract.

```
pub enum ExecuteMsg {
    ....
    RenounceControl{},
    RenounceIssuance{},
    TransferOwnership{
        new_owner: String,
    },
    RenounceOwnership{},
    ....
}
```

All these execute messages can only be executed by the Register Administrator.

Finally, we have the execute functions related to the management of constraints as stated in *Art. 9: Establishment of constraints*.

Add Constraint

In particular, this message helps us to add a constraint to a financial instrument indicating some critical data.

```
pub enum ExecuteMsg {
    ....
    AddConstraint{
        id: String,
        token_holder: String,
        recipient: String,
        beneficiary: String,
        amount: Uint128,
        ctype: String,
        is_blocking: bool,
```

```
        meta: String,
        expiration: Expiration,
    },
    ....
}
```

Specifically, the message's fields are intended as follows:

- *id* → a unique identifier of the constraint
- *token_holder* → the owner of the tokens the constraint is about to be applied on
- *recipient* → the address of the recipient of the constraint
- *beneficiary* → the constraint beneficiary that, in some cases, can differ from the recipient
- *amount* → the amount of tokens to constraint
- *ctype* → the type of constraint. For instance, a constraint can be a "pledge" or an "usufruct" and so on.
- *is_blocking* → this field is a boolean that state if the constraint is blocking or not. If the constraint is blocking, the constrained tokens are locked and cannot be transferred or sent to other wallets.
- *meta* → used to attach metadata to the constraint. It can also store a URI instead of storing directly the metadata.
- *expiration* → indicates an eventual expiration time of the constraint. The expiration time must be in the future.

Remove Constraint

This execute message, instead, is removing a constraint from the financial instruments.

```
pub enum ExecuteMsg {
    ....
    RemoveConstraint{
        token_holder: String,
        id: String,
    },
    ....
}
```

In particular, the sender must indicate the unique id of the constraint and the owner of the tokens.

update Constraint

To update a constraint, we need to indicate the unique id of the constraint and the owner of the tokens.

```
pub enum ExecuteMsg {
    ....
    UpdateConstraint{
        id: String,
        token_holder: String,
        recipient: Option<String>,
        beneficiary: Option<String>,
        amount: Uint128,
        is_blocking: bool,
        meta: String,
        expiration: Expiration
    },
    ....
}
```

Furthermore, the *update* message has the same structure as the *add* message except for the optional *recipient* and *beneficiary* fields. In addition, once set, the nature of a constraint can no longer be changed, which explains the absence of the *ctype* field.

Also in this group of execute messages, only the Register Administrator and the Issuer have the privileges to execute them.

Essentially, these are the main execute messages implemented to provide a security token for the tokenization of financial instruments and compliant with the regulatory framework described in the first chapters.

Query messages

As stated in *Art. 4: Requirements of the Registers for Digital Circulation*, all the information stored in the contract must be available. Thus, the following queries are meant to return, each of them, a specific set of data.

As mentioned before, a query does not change or delete data, it only retrieves them if available; therefore, there are no privileges and every connected user can

query and view this data.

```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult<Binary> {
    match msg {
        QueryMsg::TokenInfo{} => to_json_binary(&query_token_info(deps)?),
        QueryMsg::Controllers {} =>
            to_json_binary(&query_controllers(deps)?),
        QueryMsg::DefaultPartitions{} =>
            to_json_binary(&query_default_partitions(deps)?),
        QueryMsg::IsControllable{} =>
            to_json_binary(&query_is_controllable(deps)?),
        QueryMsg::IsIssuable{} => to_json_binary(&query_is_issuable(deps)?),
        QueryMsg::Balance { address } =>
            to_json_binary(&query_balance(deps, address)?),
        QueryMsg::BalanceOfByPartition { address, partition } =>
            to_json_binary(&query_balance_of_by_partition(
                deps,
                address,
                partition,
            )?),
        QueryMsg::BalanceOfByAllPartitions { address } =>
            to_json_binary(&query_balance_of_by_all_partitions(
                deps,
                address,
            )?),
        QueryMsg::Minter {} => to_json_binary(&query_minter(deps)?),
        QueryMsg::MarketingInfo {} =>
            to_json_binary(&query_marketing_info(deps)?),
        QueryMsg::Owner {} => to_json_binary(&query_owner(deps)?),
        QueryMsg::Whitelist {} => to_json_binary(&query_whitelist(deps)?),
        QueryMsg::IsWhitelisted { account } =>
            to_json_binary(&query_is_whitelisted(deps, account)?),
        QueryMsg::Constraints {account, start_after, limit } =>
            to_json_binary(&query_constraints(
                deps,
                account,
                start_after,
                limit,
            )?
        ),
    }
}
```

Every query refers to an execute message described previously or to the instantiate message. In particular, each of them returns the following:

- *TokenInfo* → it returns basic informations about the token itself. If not modified, the returned data were supplied during the instantiation process.
- *MarketingInfo* → it returns the marketing informations of the token if previously provided.
- *DefaultPartitions* → this query instead, returns the list of default partitions set during contract instantiation.
- *Balance* → the token balance of the specified user address. It returns how many tokens it actually owns.
- *BalanceOfByPartition* → it returns the balance of the specified user for a specific partition. For instance, if the user has only one partition, this will coincide with the result of *Balance*.
- *BalanceOfByAllPartitions* → It returns the balance of the user for each partition it owns. For example if the user has partition A, B and C, this query will return: `[Partition: A, Balance: 3780, Partition: B, Balance: 400, Partition: C, Balance: 0]`
- *Owner* → returns the actual owner of the security token. It is worth noting that the owner is intended as the role of the Register Administrator.
- *Minter* → returns the contract minter address if set. Also here, it is important to remind you that the minter assumes the role of Issuer.
- *Controllers* → returns the list of controllers of the contract. It can also return an empty list.
- *Whitelist* → this query instead, returns the list of the whitelisted address.
- *IsWhitelisted* → returns a boolean that states if the specified account is whitelisted or not.
- *IsControllable* → returns a boolean that indicates if the contract is controllable. In particular, when we execute the *RenounceControl* message, this query will return false.
- *IsIssuable* → same as the previous query, it returns if it is possible to mint new tokens or not.
- *Constraints* → it returns all the constraints of the security token with the possibility to filter by a specific account address, limiting the number of returned constraints and/or starting listing after a specific bound.

4.0.4 Events

A key requirement by the Fintech decree for the management of financial instruments is the transparency. In the Cosmos ecosystem, there are what are called ***Events***. They allow a blockchain to attach key-value pairs to a transaction. These events can be used later to search for a transaction or to retrieve the attached information [21].

The events in the cosmos ecosystem are dispatched by both the modules and the smart contracts and they have a specific format. Basically, each event has a string type and a list of attributes.

```
{
    "type": "wasm",
    "attributes": [
        {
            "key": "_contract_address",
            "value": "did:com:1rsrefjc7xnl6d6fm6avl706nu5y6nkpxaa9
                    qnpqpzs67pk7vzjdqcup0hc",
        },
        {
            "key": "transferred",
            "value": "987654",
        }
    ]
}
```

Usually, the type field corresponds to the module name, the action performed or to the general "message" key.

As previously mentioned, smart contracts also can emit events. The events are defined in the code and dispatched as the last action in the execution flow. Additionally, in the case where the contract dispatches sub-messages and these sub-messages issue some events, they will be attached to the events of the top-level message.

Furthermore, the wasm module emits standard events for each message it processes. Basically, the module emits a "message" event and then the specific event for the executed message. For instance, in the case of a contract instantiation, this should be the emitted events from the wasm module:

```
sdk.NewEvent(
    "message",
    sdk.NewAttribute("module", "wasm"),
```

```
    sdk.NewAttribute("action", "/cosmwasm.wasm.v1.MsgInstantiateContract"),
    sdk.NewAttribute("sender", msg.Sender),
),

// Instantiate Contract
sdk.NewEvent(
    "instantiate",
    sdk.NewAttribute("code_id", fmt.Sprintf("%d", msg.CodeID)),
    sdk.NewAttribute("_contract_address", contractAddr.String()),
)
```

Concerning the custom events issued by our CW858 contract, we followed the same logic and format. Each function emits an event with the related list of attributes and the type of the event is the action being performed. So for instance, if the execution is about a *transfer* of tokens, the event type will be *transfer*. Additionally, every event contains the sender attribute in order to always track the user that executed a certain action. Finally, additional attributes, if any, are specific to each function.

In the following, some events that are issued by the contract as examples.

Event Attribute	**Value**
action	transfer
sender	{senderAddress}
to	{recipientAddress}
amount	{amount}

Table 4.1: Transfer Message Event

Event Attribute	**Value**
action	mint
sender	{senderAddress}
to	{recipientAddress}
amount	{amount}

Table 4.2: Mint Message Event

Event Attribute	Value
action	add_whitelist
sender	{senderAddress}
account	{walletAddress}

Table 4.3: AddWhitelist Event

4.0.5 eREG Backend

In the second chapter we went through the structure of the eReg platform giving an overview of the main components and also describing the interaction between the components.
After initial configuration, our main focus is on the Backend of eRreg.

As introduced previously, the backend is the entity between the user and the blockchain. Its main objective is to facilitate the interaction with the blockchain where our CW858 is deployed. Additionally, it also represents a fundamental component for the implementation of a platform compliant with the Fintech decree since it enforces various statements before interfacing with the blockchain.

Essentially, the backend is implemented in **Golang** and designed to follow the **Layered Architecture** pattern, also known as the *N-tier Architecture*.

This architecture design divides the application into several distinct layers where each layer has a specific task. The layers are independent of each other and can directly interact but only with the adjacent layers.

Furthermore, in the N-tier architecture, there are four layers. Specifically:

- **Presentation Layer**:

  This layer is responsible for interfacing with the user. It handles user inputs, calls the business layer for computation and format data to return to the user.

- **Business logic layer**:

  The Business logic or application layer instead, is core component of the architecture and is responsible for processing the user requests and executing business rules.

- **Persistence Layer**:

  This layer is used by the business layer to handle data on the database. It contains functions to facilitate data update or data retrieval to or from the

database allowing more abstraction. However, in some layered application this layer may be embedded inside the business logic layer.

- **Database layer**:

  Finally, this last layer is were the application actually stores its data.

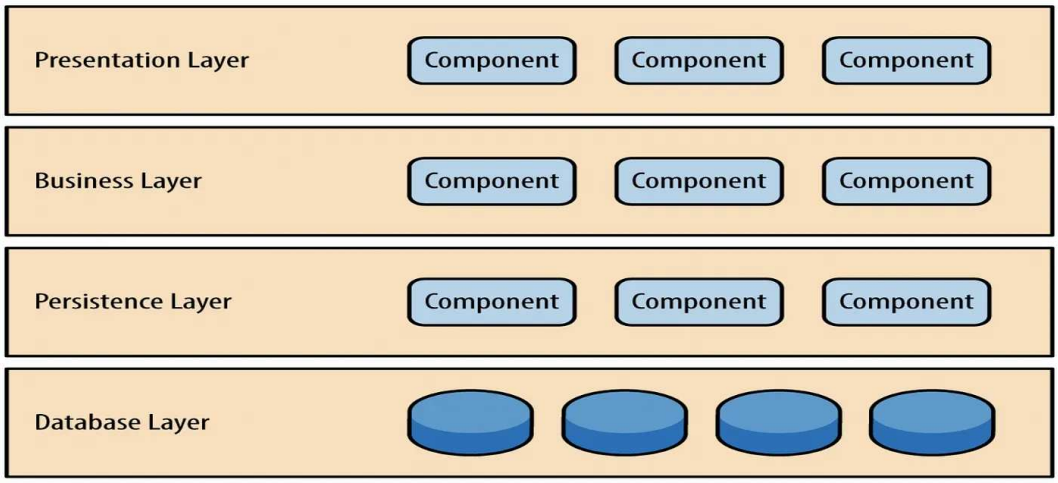

Figure 4.2: Layered Architecture [22]

Furthermore, there are several advantages of following this design pattern. The main advantage is the separation of concerns, having each layer responsible for different tasks makes the application easier to maintain and modify. Another advantage is the testability, since each layer can be tested separately mocking the adjacent layers. Additionally, the ease of development is another point of benefit of this pattern.

However, the high coupling of the layers makes the application difficult to change and time-consuming. Also, having multiple layers can lead to low performance since we must go through many layers to process and return the data back to the user.

REST API

In our backend application, the REST handler represents our presentation layer since it handles all the HTTP requests and responses. In particular, we defined a set of APIs to enable a user to interact with the blockchain and thus, the deployed smart contract.

Figure 4.3: APIs

The fig 4.3 is generated with the help of ***Swagger*** and besides describing all the implemented endpoints of the backend, it gives us the possibility to test them with the *"try out"* feature.

Furthermore, as the figure 4.3 shows, the endpoints are self-describing in respect to the CW858 messages introduced earlier.

Essentially, the body of these endpoints includes all the necessary data that needs to be provided to successfully set the related CW858 message. Additionally, in some endpoints there are extra fields to be included, for instance: the address of the CW858 instance to execute.

Business logic

As described previously, this is the core layer of our backend application. In fact, all the HTTP requests handled by the previous layer are processed in this component where all the business logic is applied.

Essentially, every request is processed following a quite general sequence of operations.

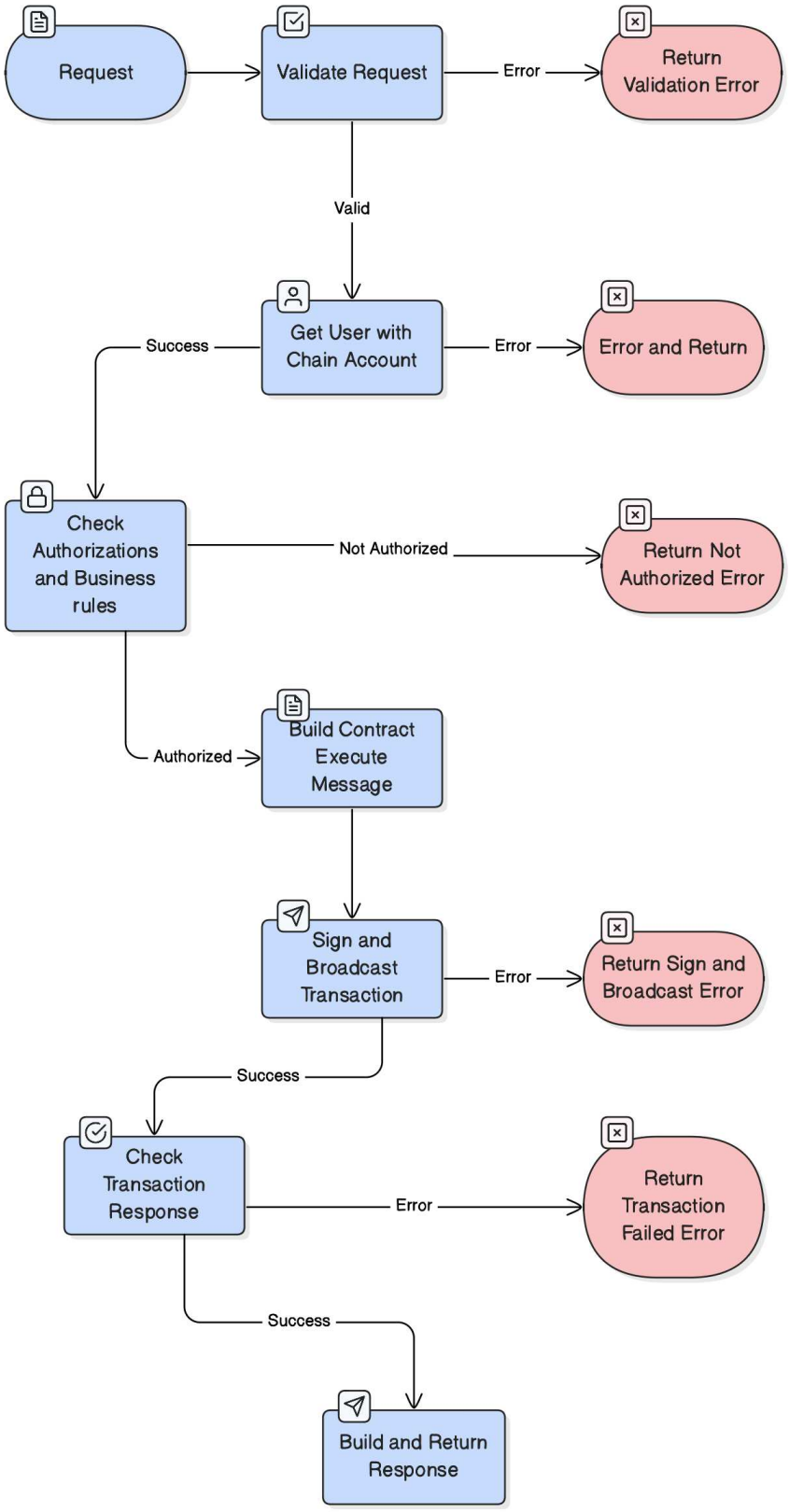

Figure 4.4: Controller flow

As shown in the flow chart 4.4, the first action is to validate the request body forwarded from the REST handler, this to ensure that all the mandatory data are provided and in the right data type. Successively, we retrieve the user's wallet address using its access token. The next step concerns the business logic to be applied for the called function.
At this stage, if all the previous steps were successful, the CW858 message is built, signed with the user's private key and broadcasted to the blockchain. Once the contract is executed on the blockchain, the transaction response is returned and from the successful response we return the transaction hash to the handler.

Persistent and Database Layer

These two layers are strictly related since the persistent layer can be seen as a function wrapper of the database layer. In our eReg application, these two layers are almost absent because ***the blockchain is our database***.

Thanks to all the properties of the blockchain described in the previous chapter, it allows us to store all the necessary informations related to the management of financial instruments in it.

In conclusion, both the contract and the backend play the core function of this project and only when combined together they can fulfil the regulatory framework requirements and the desired usability of the system.

Chapter 5

Tests and Results

5.1 Tests

During the implementation of this project, the tests were conduced in different phases and with different scopes.

Initially, the CW858 smart contract has been implemented and tested along with its development, then the backend has been implemented and tested. Finally, the whole system has been connected and tested all together.

In general, the tests are executed within two different environments. I have configured a local chain of the Commercio.network on my computer where I deploy the smart contract, interact with the chain's functions and execute all the tests. Then on the ***Devnet***, the Commercio blockchain we use for functionality testing during development, more tests are executed from other members of the company.

5.1.1 CW858 tests

Furthermore, for the tests of the smart contract, the consequent steps were followed:

- Deploy on the local chain the new version of the smart contract
- Test the functionalities and the business rules
- Merge request to the *develop* branch
- release the new code and deploy the new smart contract on the *Devnet*
- Further tests from the chain's users

The whole procedure is performed for each singular message implementation, so for instance, it is performed for the instantiate message and mint message.

5.1.2 Backend tests

Since the Backend interacts directly with the blockchain, HSM and the Keycloak, it is tested differently from the smart contract.

In particular, a **Docker** stack is set with the following containers:

- Keycloak
- Postgres Db for Keycloak
- Postgres Db for the backend
- Foxhsm
- Backend

Each time a new functionality is implemented in the backend, a new image is built and the stack is updated with the new container. On the other side, the local commercio.netwrok chain is run and the final version of the smart contract is deployed in it.

Locally the tests are run interacting with the backend through **Postman** where we actually set the API URL to request, the Bearer Token for authorization and eventually the Body of the request.

Also in this phase, the tests are run on remote by the other developer of Commercio.network. In particular, on a specific virtual machine the same Docker stack is run and communicates directly with the Devnet.

During the testing of the Backend, the CW858 smart contract is tested again both locally and remotely.

5.2 Results

With the implementation of our CW858 smart contract and the implementation of our eREG Backend, we have achieved the goal of building one of the first, if not the only, platforms for the tokenization of financial instruments and compliant with the regulatory framework.

With the support of our blockchain, many of the properties required by the regulatory are satisfied by design. The CW858 is built from a standard smart contract for security tokens, specifically the ERC1400 from Ethereum and the Backend is based on secure frameworks for Authentication and Authorization while handling specific business rules before interfacing with the other components of the infrastructure.

Furthermore, our platform enables a Register Administrator or an Issuer to fully operate with the financial instruments without having to worry about how a blockchain or a smart contract works while leveraging their advantageous properties.

In detail, one is able to tokenize a financial instrument by instantiating a new CW858 contract with the required information. The Issuer is also able to issue new financial instruments or remove them from the circulation. Additionally, one can transfer the financial instruments or add a constraint to their circulation.

In the following, some screenshots of the implemented Frontend that interfaces with the Backend to add more flexibility and usability to the system.

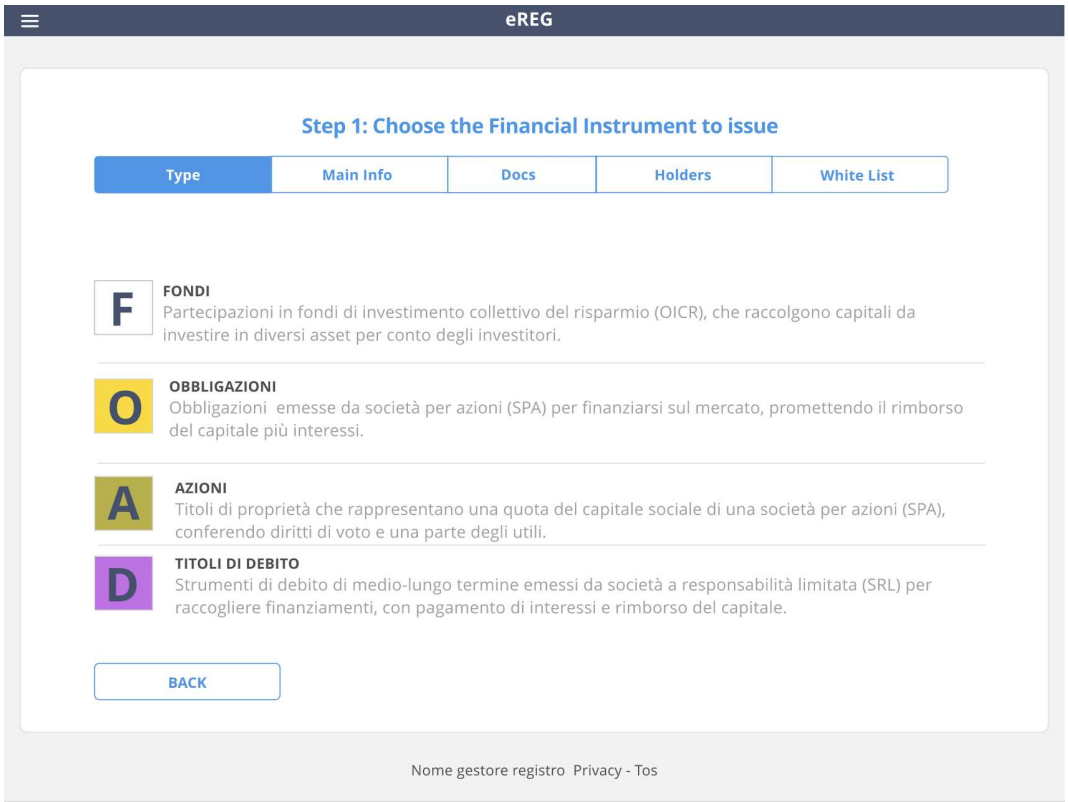

Figure 5.1: Type of financial instrument to issue

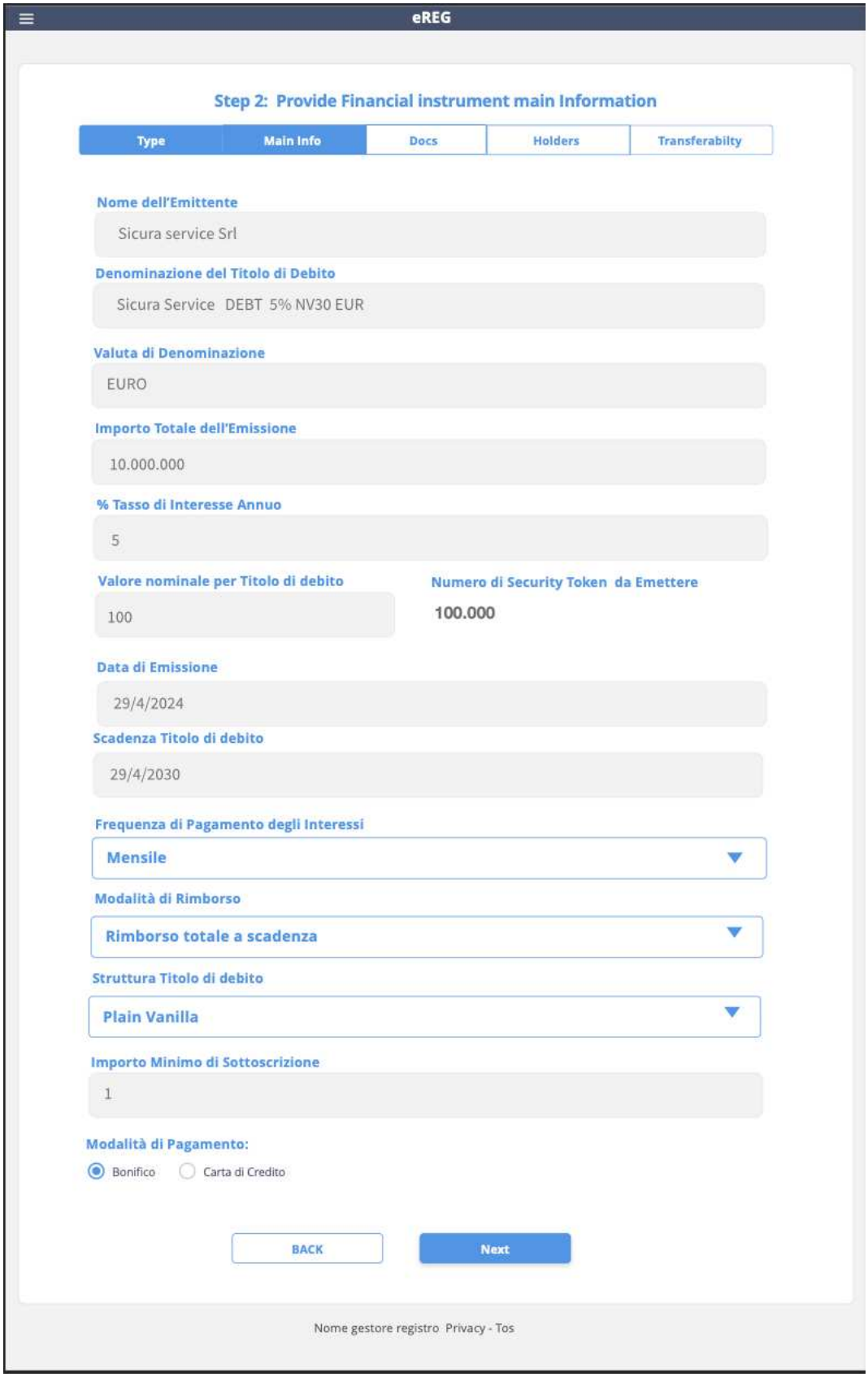

Figure 5.2: Main informations

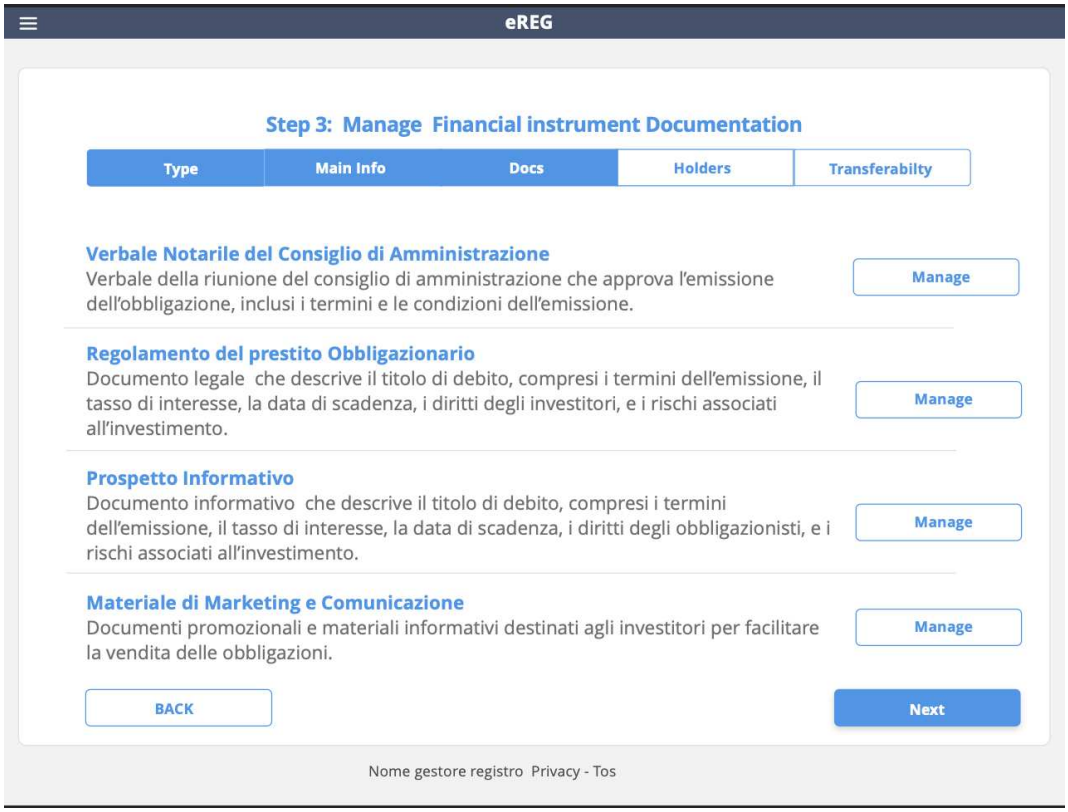

Figure 5.3: Official documentation to attach

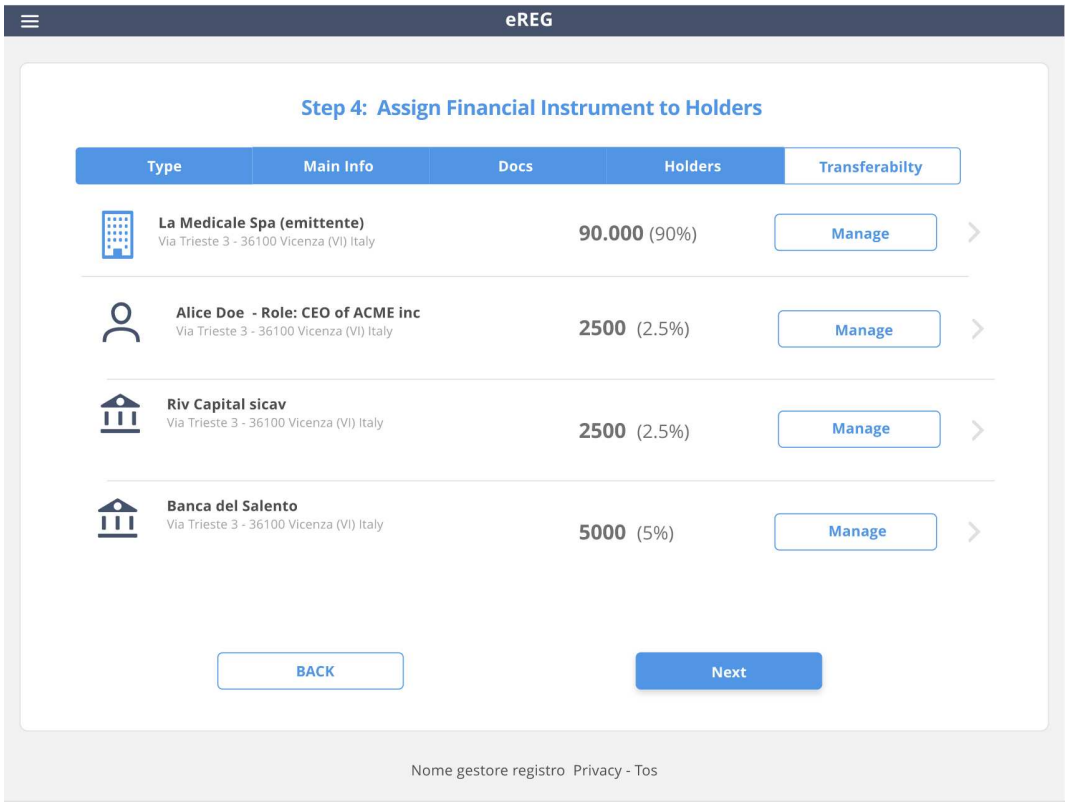

Figure 5.4: Initial token holders

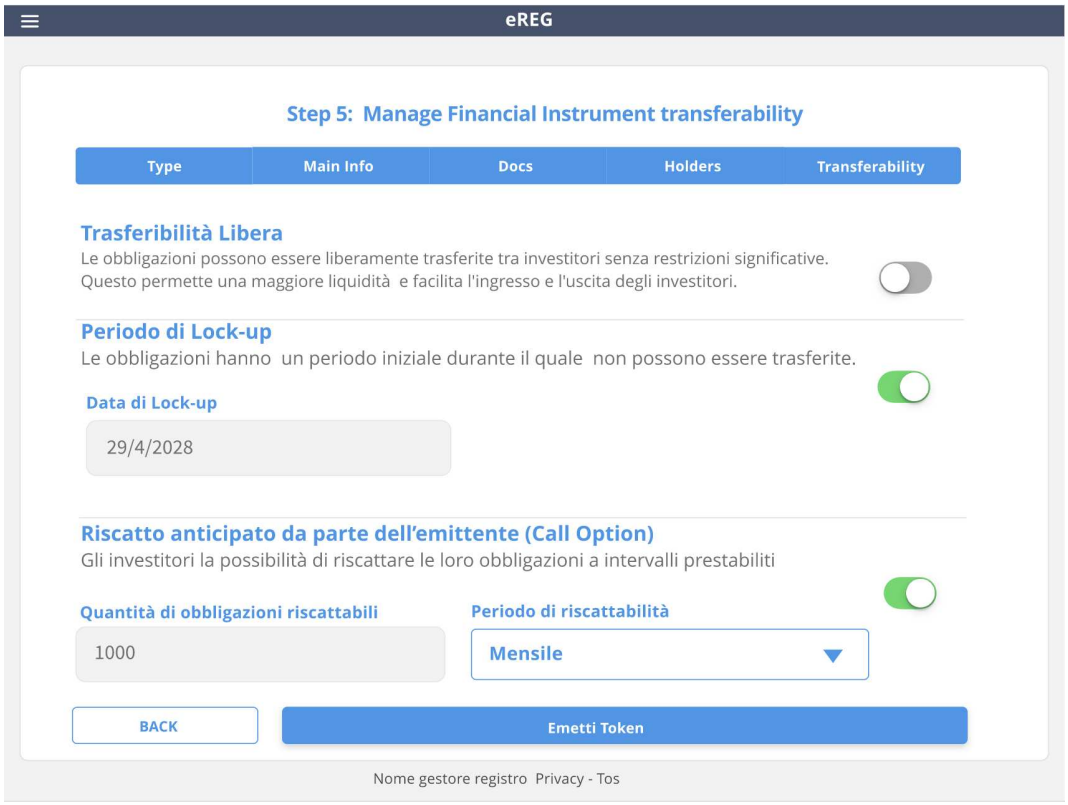

Figure 5.5: Final business rules

The figures 5.1, 5.2, 5.3, 5.4 and 5.5 represent the 5 established steps for financial instrument issuance.

Moreover, similar steps and procedures are provided for the remaining functionalities offered by the Backend.

Chapter 6

Conclusions

In conclusion, this project was the intersection of three worlds: the regulatory world, the finance world and the tech world. As a Software engineer, it was challenging to face and actually understand the regulatory framework and the financial instruments functioning to be able to correctly interpret the requirements and implement a compliant system.

The project was born with the continuous integration of the blockchain in software applications and with the come into effect of the Fintech Decree that opens new opportunities for all kinds of companies. There are several advantages in tokenizing financial instruments, as described along the thesis, that foster companies like Commercio.network to undertake this path with all the annexed challenges.

Supported by a team of lawyers, we extracted the key information from the Fintech Decree and modelled it for the implementation of the CW858 smart contract in the first place and then the eREG Backend.

Moreover, before all the work, there was a study and research time where we dug into the ERC1400 standard, the EU regulation and the Backend infrastructure design.

Finally, the just overviewed project is our **MVP**[1] to submit to CONSOB for the permission request to be able to legally operate within the regulatory Sandbox.

Engaged future works include the following:

- obtain an authorization from CONSOB
- after authorization proceed to the next steps of testing the infrastructure by deploying to *Testnet* and then finally release to the *Mainnet*.

[1] MVP stands for Minimum Viable Product

- automate action execution on constrained tokens based on the type of constraint. For instance, we suppose that Alice has x tokens with a *pledge* constraint and Bob is the beneficiary, we should automate that when certain required conditions are met, the contract automatically transfers these tokens from Alice to Bob.
- study the requirements and implement a *marketplace* to buy and sell financial instruments. Actually, the users are able to transfer tokens to each other after an agreement out of the platform.
- study the requirements and implement the Burn of tokens in the Backend side

Overall, the system is ready to be tested in a real world scenario and successively be released as the first tokenization platform of financial instruments in Italy. Finally, the project is implemented to grow and evolve based on the outcomes of this sandbox.

Bibliography

[1] Italia Fintech. “Decreto Fintech n. 25 del 17 marzo 2023”. In: (Apr. 2023) (cit. on pp. 6, 7).

[2] IBM. *Blockchain*. Jan. 2025. URL: https://www.ibm.com/think/topics/blockchain (cit. on p. 11).

[3] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (cit. on p. 11).

[4] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. *An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends*. 2017. URL: https://arxiv.org/abs/1708.04873 (cit. on p. 11).

[5] Investopedia. *Consensus Mechanism (Cryptocurrency)*. Accessed: 2025-02-10. n.d. URL: https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp (cit. on p. 12).

[6] Techskill Brew. *Types of Blockchain - Part 4: Blockchain Basics*. n.d. URL: https://medium.com/techskill-brew/types-of-blockchain-part-4-blockchain-basics-c0b2e40c1780 (cit. on pp. 15, 17).

[7] Enrico Talin. *Fare business con la blockchain*. First Edition. Schio (VI): Tradenet Services Srl Editore (IT), 2019 (cit. on p. 16).

[8] Mayuko Kondo. *EMURGO Africa 2023 Q1 Report Ch. 2.1: Types of Tokens on Blockchain*. 2023. URL: https://medium.com/@mayuko.kondo/emurgo-africa-2023-q1-report-ch-2-1-types-of-tokens-on-blockchain-998e0ad4a979 (cit. on p. 19).

[9] Dapper Labs. *Dapper Labs Official Website*. n.d. URL: https://www.dapperlabs.com (cit. on p. 19).

[10] Commercio Network. *The White Paper*. 2023. URL: https://commercio.network/it/progetto/ (cit. on p. 20).

[11] Cosmos Network. *Cosmos SDK Overview*. [Accessed: 15-02-2025]. 2024. URL: https://docs.cosmos.network/v0.45/intro/overview.html (cit. on p. 23).

[12] IBC Protocol. *IBC Protocol Documentation*. Accessed: February 2025. 2024. URL: https://ibcprotocol.dev (cit. on p. 25).

[13] Red Hat. *Keycloak Documentation*. 2024. URL: https://www.keycloak.org/documentation (cit. on p. 35).

[14] OpenID Foundation. *OpenID Connect Core Specification*. 2021. URL: https://openid.net/specs/openid-connect-core-1_0.html (cit. on p. 36).

[15] Internet Engineering Task Force (IETF). *JSON Web Token (JWT) Profile for OAuth 2.0*. 2015. URL: https://tools.ietf.org/html/rfc7519 (cit. on p. 36).

[16] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. Available at: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. 1996 (cit. on p. 41).

[17] Investopedia. *Smart Contracts Definition and Overview*. [Accessed: 17-Feb-2025]. June 2024. URL: https://www.investopedia.com/terms/s/smart-contracts.asp (cit. on pp. 41, 42).

[18] Ethan Frey. *CosmWasm for CTOs I: The Architecture*. [Accessed: 26-02-2025]. Oct. 2021. URL: https://medium.com/cosmwasm/cosmwasm-for-ctos-i-the-architecture-59a3e52d9b9c (cit. on p. 42).

[19] CosmWasm Team. *CosmWasm Entry Points*. [Accessed: 17-Feb-2025]. 2024. URL: https://cosmwasm.cosmos.network/core/entrypoints (cit. on p. 43).

[20] World Federation of Exchanges & ConsenSys. *Introduction to Tokenization*. [Accessed: 17-Feb-2025]. 2024. URL: https://focus.world-exchanges.org/articles/introduction-tokenization-consensys (cit. on p. 47).

[21] CosmWasm. *WasmD Events Documentation*. [Accessed: 03-Mar-2025]. 2025. URL: https://github.com/CosmWasm/wasmd/blob/main/EVENTS.md (cit. on p. 59).

[22] Mark Richards. *Software Architecture Patterns*. First Edition. Sebastopol, CA: O'Reilly Media, Inc., 2015. URL: https://isip.piconepress.com/courses/temple/ece_1111/resources/articles/20211201_software_architecture_patterns.pdf (cit. on p. 62).