ChainVote Final Report

(v. 1.0.0)

Giovanni Antonioni
giovanni.antonioni2@studio.unibo.it

Luca Rubboli
luca.rubboli2@studio.unibo.it

Luca Tassinari
luca.tassinari10@studio.unibo.it

December 6, 2023

Electronic voting systems based on blockchain technology have emerged as a potential solution to enhance the security and transparency of traditional voting methods. In this system, voters cast their votes electronically, and the results are stored on a decentralized blockchain ledger, which ensures the integrity of the vote by preventing any tampering or manipulation. This system provides a transparent and immutable record of votes, which can be accessed by anyone in the network, thus increasing trust in the electoral process. Nevertheless, the use of blockchain technology in electronic voting systems holds promise for creating a more secure, transparent, and democratic electoral process.

1 Goals/requirements

The project consists of the implementation of a small-scale distributed electronic voting system based on blockchain technology. Specifically, the goal is to create a distributed architecture that exposes a uniform API allowing users to interact with the system using a web application, as described in the use scenarios presented in Section 1.1.

Here are reported the set of questions and answers used to refine the system's requirements.

Q1. What are the main actors of the system?

A. Two main actors can be identified in the system: the **administrator**, taking care of creating new elections, and the **user**, which interacts with them.

Q2. Can everybody create a new election?

A. No, only system administrators can create new elections.

Q3. Can everybody cast votes?

A. System administrators can't cast a vote, while regular users are allowed to cast a vote only once authenticated.

Q4. How many times can a user cast votes in a single election?

A. A user can cast a vote up to once per ballot.

Q5. What is an election?

A. An election is a set of mutually exclusive choices with a time limit that cannot be changed.

Q6. Can a user modify their vote?

A. No, once a vote has been cast it cannot be modified.

Q7. Which information about the election is available to users?

A. While a vote is open users can see only the turnout. Once an election is closed, all users (voters and abstainers) and administrators can see the results. At the application level, the connection between a user and his vote must be hidden.

Q8. Is it possible to have multiple elections open at the same time?

A. Yes.

1.1 Scenarios

In Figure 1 are reported the main use cases that emerged from the requirements analysis.

1.2 Self-assessment policy

Assessment will be conducted through automated tests, ensuring that all the requirements identified in the analysis will be thoroughly covered.

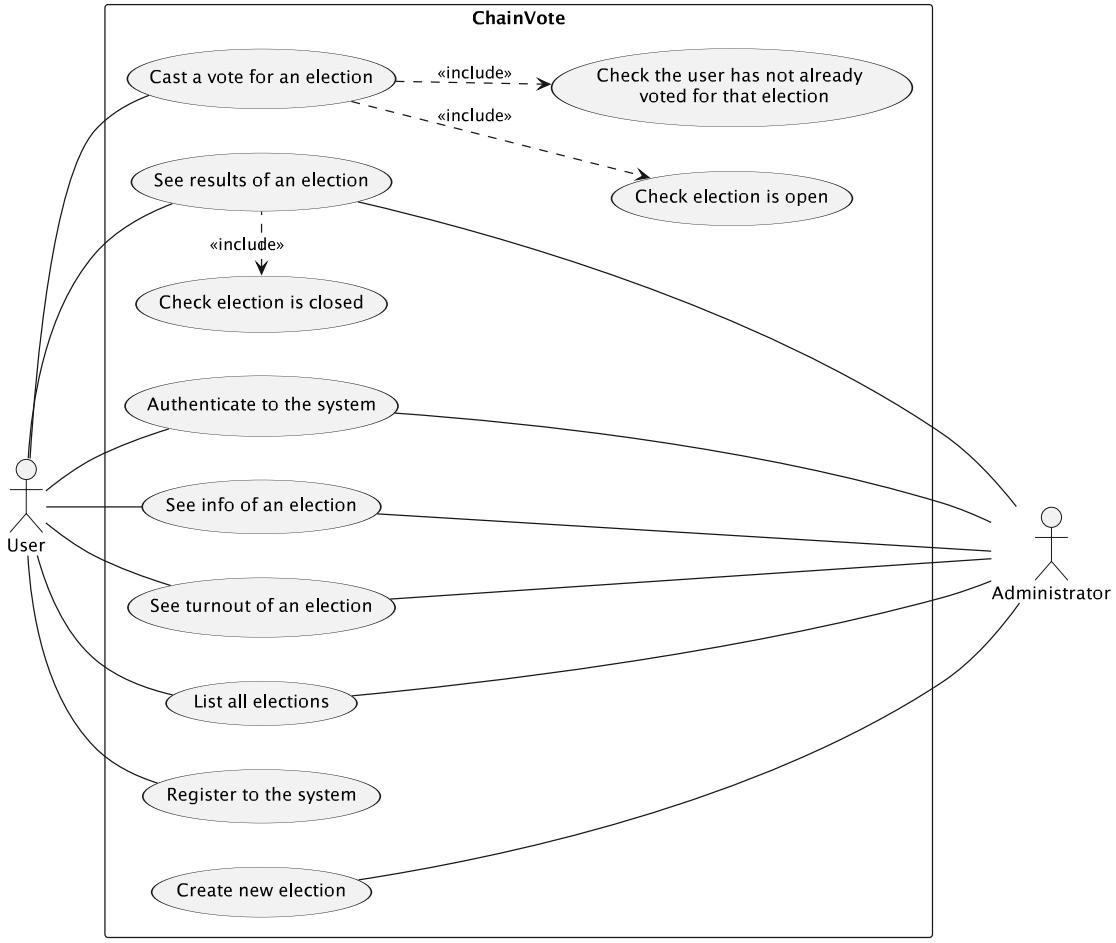


Figure 1: Use cases diagram of the ChainVote application.

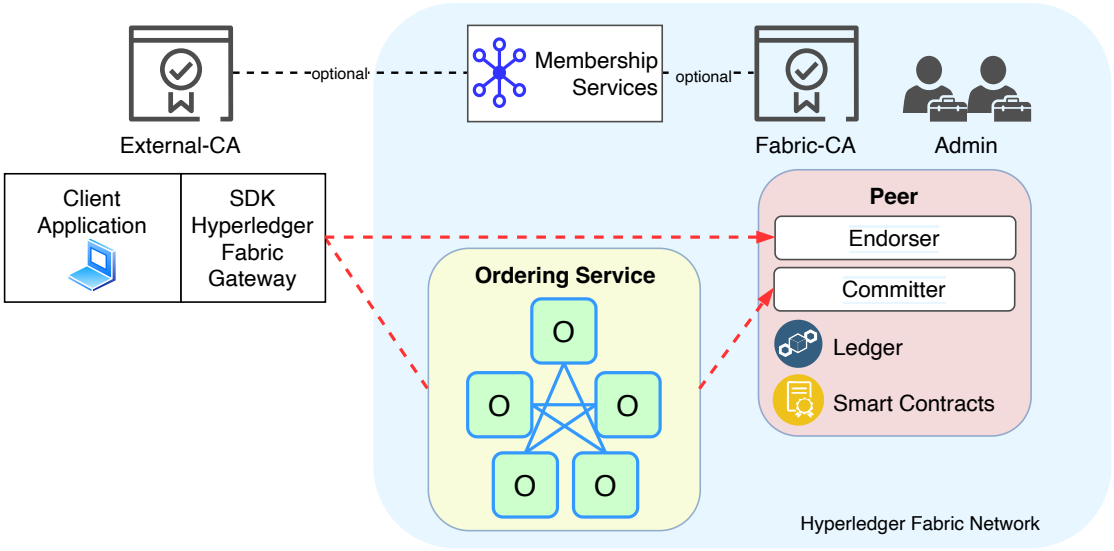


Figure 2: Hyperledger Fabric main components [6].

2 Background

2.1 Hyperledger Fabric

Hyperledger Fabric (HLF) stands as one of the leading open-source [3] blockchain frameworks specifically created in 2015 by the Linux Foundation for enterprise applications. This framework offers organizations tools and a highly modular and configurable architecture necessary to build permissioned blockchain networks, making it ideal for projects seeking to harness the benefits of distributed ledger technology (DLT) while maintaining control and confidentiality over their data.

Before delving into the key concepts of this technology we have to distinguish **permissioned** over **permissionless** blockchains. In the latter anyone can participate, participants are anonymous and trust primarily relies on the immutability of the distributed ledger. Such blockchains typically use a byzantine fault tolerant consensus protocol based on *proof of work (PoW)* end economic incentives to compensate the costs for participating in it. In contrast, permissioned blockchains are closed: they have an access control layer ensuring participants, who are identified by digital certificates, perform only the actions they are authorized to perform, fostering a higher level of trust through a governance model. This model makes it possible, on the one hand, to use more traditional fault-tolerant Byzantine consensus protocols that do not require expensive mining and, on the other hand, to provide an additional layer of security.

HLF components overview

The main Hyperledger Fabric infrastructural components are depicted in Figure 2.

- The **membership service** component is in charge of managing network participants identities. The Hyperledger Fabric CA is a possible certificate authority-based implementation of membership service directly provided by Hyperledger Fabric, but any Public Key Infrastructure that can issue certificates can be used.
- The **ordering service** is a set of dedicated orderer nodes which:
  - order and batch submitted transactions in blocks;
  - sign each batch block to create the ledger hash chain;
  - broadcast newly created blocks.

  Practically, it represents the **consensus** system. The modular structure of the Hyperledger Fabric network allows different implementations of the ordering services to be used. The Raft [11] implementation is the recommended one (even if is possible to develop a custom ordering service if desired).
- **Peers** are responsible for maintaining the ledger and the smart contracts: *endorser* ones are responsible for executing and simulating transactions requested by clients, *committers* are in charge of validating them and updating the ledger.

Organizations

An organization, in Hyperledger Fabric, defines a logical grouping of the participants of the network based on their business relationships. It provides a way to define an identity for each of its members, using a Certificate Authority (CA), for issuing certificates and an internal Membership Service Provider (MSP) that provides credentials and roles for participating in the network. Multiple organizations can interact with each other by means of a channel, i.e. a blockchain overlay on which transactions are performed on a specific ledger and regulated by a set of policies agreed by the organizations participating in the channel.

Channels

A channel is a private communication subnet for confidential transactions between specific network members. A channel is defined by one or multiple organizations, peers and order nodes, a shared ledger and chaincode applications. Transactions are executed and validated inside the channel from authenticated peers and are visible only to the organizations that participate in the channel.

Ledger

In general, a blockchain system comprises multiple nodes (peers) each of which has a local copy of the ledger that contains the current state of a business structured as a sequential log of cryptographically interlinked immutable blocks. Indeed, the utilization of hashing and liking techniques enhances the security of ledger data.

Given that querying the current state of an asset would necessitate traversing the entire transaction log, which could be computationally expensive and time-consuming, Hyperledger Fabric offers also the World State abstraction, which represents the current

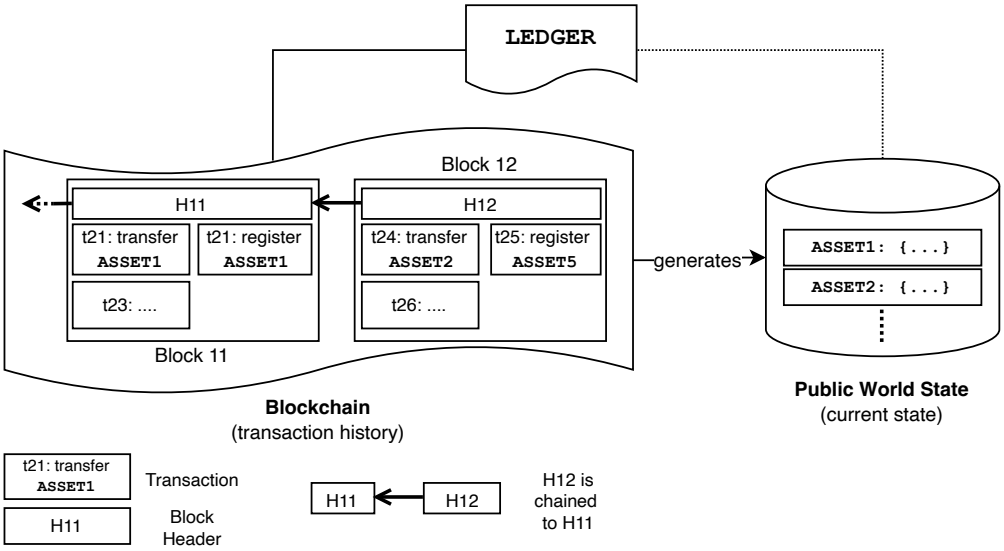


Figure 3: The Hyperledger Fabric ledger abstraction: the blockchain and the derived world state [6].

state of the blockchain at any given point in time. Physically, it is implemented as a document database that holds the current value of all the assets and data. Currently are supported LevelDB [2] and CouchDB [1]: the first is the default and is particularly appropriate when ledger states are simple key-value pairs; CouchDB, instead, is a particularly appropriate choice when ledger states are structured as JSON documents because it supports rich queries.

The overall Hyperledger Fabric ledger abstraction is summarized in Figure 3.

Private Data

Voting mechanisms need policies for saving sensitive information to maintain voters' anonymity and privacy. Inside Hyperledger Fabric this can be achieved in multiple ways: a naive approach would be to create different channels composed of a subset of the organizations that could access some specific data, but this would lead to a proliferation of channels and a consequent increase in the complexity of the network. Another approach could involve the usage of private data collection. This mechanism enables the share of private data between authorized peers of an organization using a private state database. Data remains available only to some authorized entities while a hash is sent to the ordering service as evidence of the transaction

Smart contracts

A smart contract is a set of rules that defines the transaction logic that controls the lifecycle of each business object contained in the world state; related smart contracts

can be grouped in a chaincode to provide an easier deployment to a blockchain network. A smart contract may access two distinct parts of the ledger:

1. The blockchain, which immutably records the history of all transactions;
2. The world state, that holds a cache of the current values of the states.

Smart contracts rely on 3 main actions to query the blockchain and to interact with the world state:

- `GET`, which represents a query to retrieve information;
- `PUT`, which creates or updates a business object in the current world state;
- `DELETE`, which removes a business object from the current state but not its history.

It's worth mentioning that the **purpose of the smart contracts** is to **generate** a **transaction response** which contains the state of the ledger before and after the transaction execution, capturing the transactional change, **not to update the ledger**. Indeed, a smart contract cannot do this unilaterally, but the whole network must reach a consensus before the ledger is updated. In other words, the smart contract says what it *would like* to occur, not what has occurred or will for sure occur.

A very important aspect to remark it's the **chaincode execution model**: chaincode's transactions are executed in **any order**, **possibly** even **in parallel**. This implies that no assumptions can be made about their execution order and their internal logic must be fully deterministic.

During the deployment of the chaincode, it's possible to define an **endorsement policy**, managing which organizations in a blockchain network must sign a transaction, generated by one of the smart contracts inside the chaincode, to be considered valid. All transactions (valid or invalid) are added to the ledger, but only the valid ones are executed, updating the world state.

Lifecycle

The following presents the transaction submission lifecycle (a diagram describing the flow is shown in Figure 4).

1. The application generates a signed transaction proposal, including the private key-signed transaction inputs;

2a-3a. This proposal is sent to the network endorsement peers according to the endorsement policy, which **simulates** the execution of the smart contract transaction writing into the response the state of the world before and after is applied: at this point, the ledger is not updated yet;

2b-3b. Responses are sent back to the SDK;

4. The SDK can now assemble each response in a multi-party transaction. Note: each response *should* be the same;
5. The multi-party transaction is sent to the ordering service and the consensus process starts!

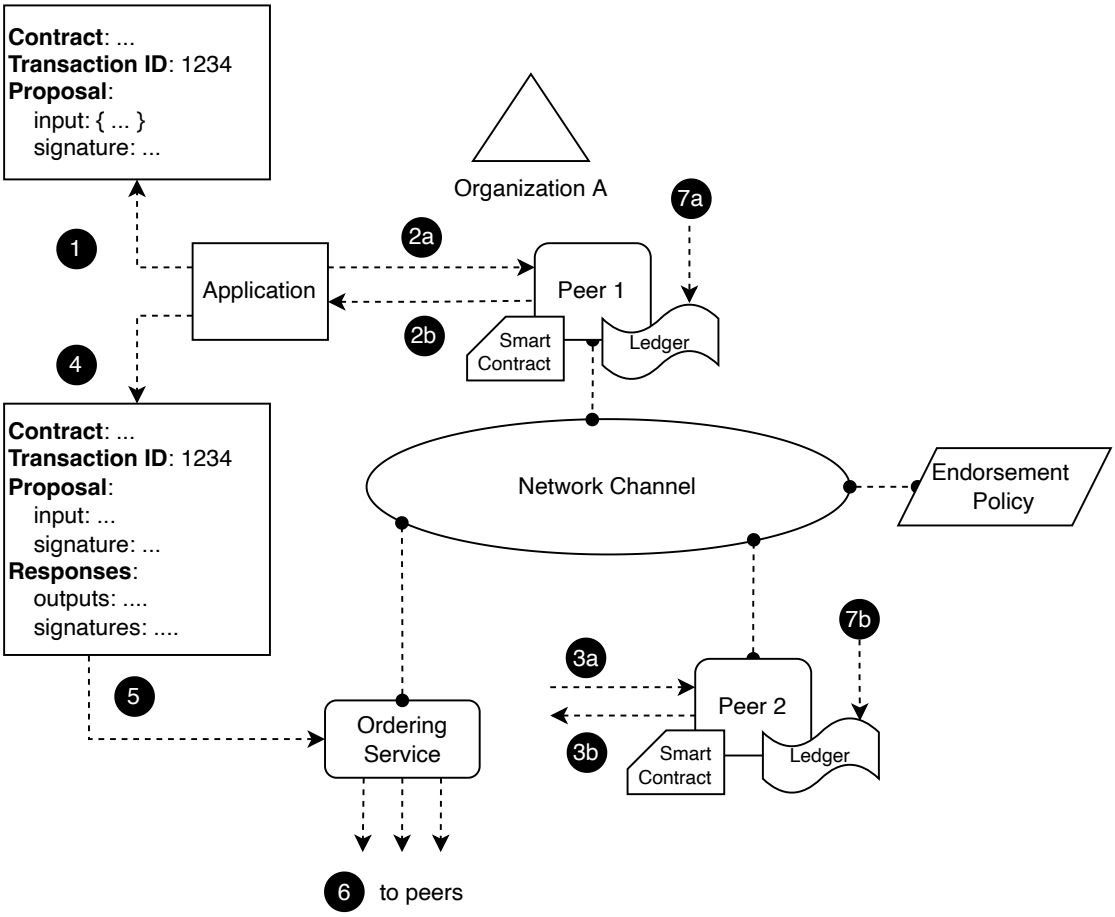


Figure 4: Hyperledger Fabric transactions lifecycle [6].

6. The ordering service packages incoming transactions in blocks and distributes them to every peer in the network;

7a-7b. The blocks are processed by each peer and valid transactions are used to update the ledger. In particular, transactions to be considered valid must be consistent with the current state of the ledger and be signed by the required number of counterparties, as specified in the endorsement policy. Consensus has been achieved when this step has been completed on all nodes. Subsequent transactions will use the new state of the ledger to generate the transaction responses.

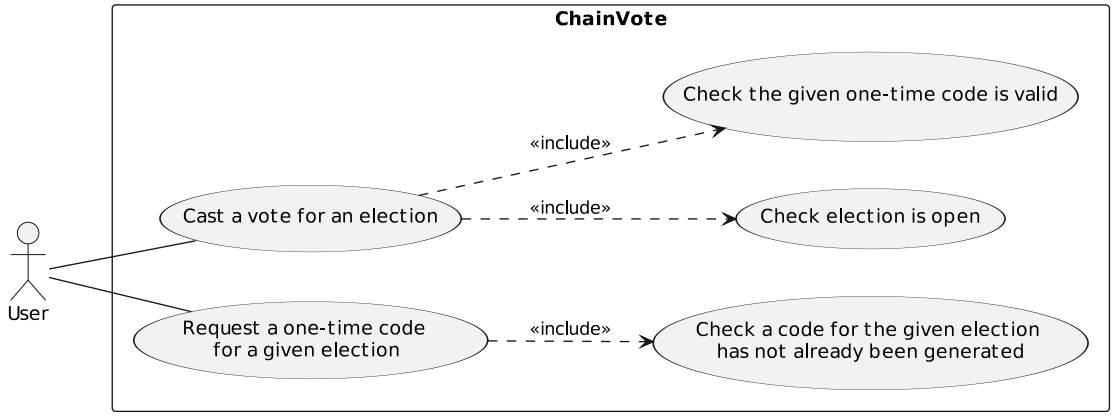


Figure 5: Refined use case diagram for the casting of a vote.

3 Requirements Analysis

3.1 Non functional requirements

In order to decouple as much as possible users from their vote inside the blockchain and make it more difficult for unauthorized individuals to cast a vote impersonating someone else, pseudo-random one-time codes will be used. The idea is that, before casting a new vote, the user requests a one-time code that is sent to him using a different trusted method. When voting, the user will have to send back the generated one-time code to the system to correctly submit his vote. The outline of use cases related to the casting of a vote in light of the above refinements is presented in Figure 5.

To accomplish this, two challenges must be faced during the design process:

- It is necessary that, within the ledger, the code-user association, as well as the expressed vote and user's code, are stored securely, i.e. are not recorded in clear in the blockchain bocks' transactions;
- Given that the deterministic nature of smart contracts makes it difficult to generate completely random codes, a deterministic algorithm will be used and a source of randomness will be injected from a trusted and controlled entity. It's worth mentioning their generation *a priori* (i.e. when the election is created) is not feasible since smart contract transactions are executed by peers in any order, possibly even in parallel, implying that different peers may process the same client transaction with different order. This means that it is not possible to guarantee that, when the client triggers the transaction execution, peers will get the same code from the set of already generated ones.

3.2 Technologies

After evaluating different valid blockchain technologies, including Corda (R3) [12] and Tendermint [13], Hyperledger Fabric has been chosen for this project for the following

main reasons:

- It is a general purpose *permissioned* blockchain, with all the advantages described in Section 2;
- It is highly configurable and customizable, like the possibility to plug different consensus mechanisms or the support of several mainstream programming languages for the smart contracts (Go, JavaScript & TypeScript, Java & Kotlin);
- It offers mechanisms that raise the bar of confidentiality and higher fine-grained control over ledger access, thanks to channels and private data collections;
- It has an active and growing community of developers and contributors, as well as detailed documentation.

In addition, one of the project goals is to provide a uniform API to interact with the system. To do this a ReST approach has been used, which is a widely adopted standard for exposing APIs. ReST gives different advantages:

- *Optimal access to the resources*: Rest requires a resource-centric design, this means that the API will be structured around the resources that are exposed. In this way, it's possible to have fine-grained control over the access and management of the resources.
- *Stateless*: ReST API requests are stateless, which means they're independent of each other. This allows to scale the system horizontally, without having to worry about the state of the system.
- *Expressive*: ReST APIs are expressive, this means that the APIs are designed in a way that is easy to understand and use. This allows to have a more maintainable and extensible system.

4 Design

The following section presents the blockchain network architecture, as well as that of the application.

4.1 Blockchain architecture

In Figure 6 is shown the blockchain network architecture that has been conceived.

In total, four main organizations are in place:

- one contains the orderer service (`org0`);
- one in charge of creating and updating the elections' static information (`org1`), like a central governance institution. This information is stored inside the ledger of channel 1;
- two participating in the voting which maintains the vote-collecting functionalities and codes management (`org2` and `org3`) by means of querying static information when needed. This information is stored inside the ledger of channel 2.

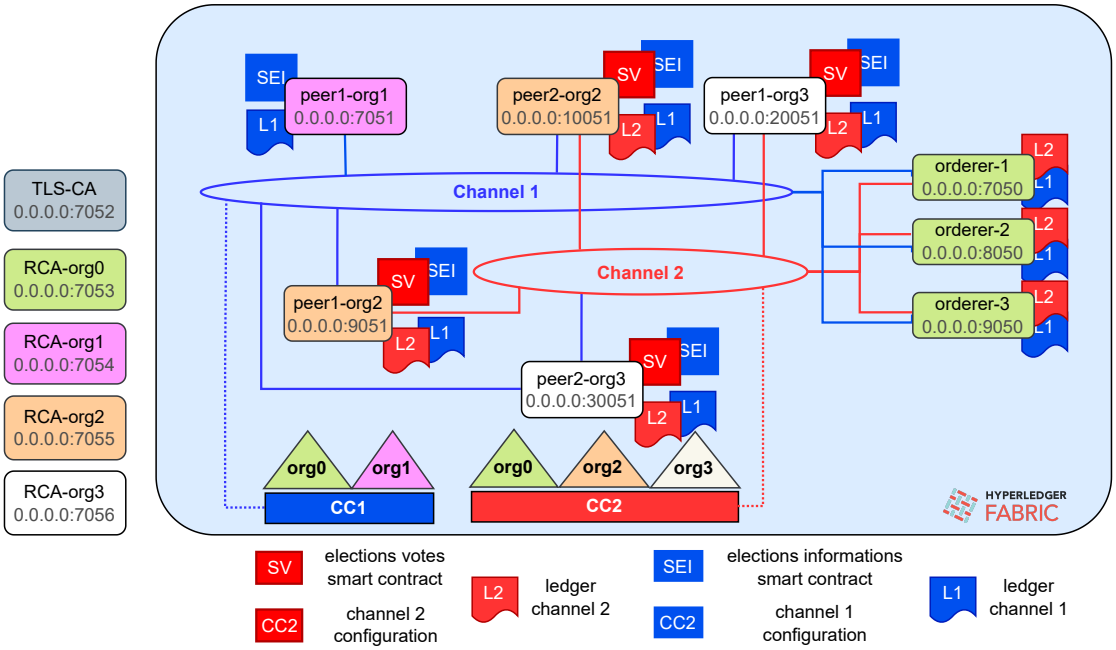


Figure 6: High-level network architecture.

For the sake of fault tolerance both `org2` and `org3` contain two peers each, while the orderer service (`org0`) contains three orderer nodes.

As already said above, as a permissioned blockchain, Hyperledger Fabric requires all the entities to be identified. This is implemented through digital certificates, managed by specific infrastructures that deal with their issuance. Those infrastructures are the certificate authorities:

- **RCA-org*x*** (x=0,1,2,3) is the Root CA for identities in each organization; it issues identity-related certificates for components and users and provides MSP functionalities during the network configuration phase.
- **TLS-CA** issuing TLS server certificates for network components (orderer and peers) for the whole network. This certification is for TLS communication only, and not related to identity in this fabric network. Concretely, when a client accesses a network component it will present its own TLS server certificate to the client. The client will use the Root CA certificate to verify the server one, and confirm the client is talking to the right server and that the communication is encrypted.

Once the certificate issuance is done and the network is set up, certificate authorities are used only in case new components or new users join the network.

Warning This is just a proof of concept of what a real blockchain network for an electronic voting system should look like. For example, it is obvious that, in a real scenario, there would be a higher number of orderer nodes allowing the system to tolerate a much higher number of failures (with the presented architecture we tolerate only one failure). The simplification is necessitated by the fact that the entire network will be deployed on a single machine using containerization tools, in which for each node (and for each smart contract) a container is created. Simulating a real blockchain network would mean running a large number of containers, which would make the deployment very computationally expensive.

4.2 Application abstract design

4.2.1 Structure / Domain Entities

Elections As shown in Figure 7, the core entities involved in election management are:

- `Election`, representing the election's vote information entity;
- `ElectionInfo`, representing the election's static information entity;
- `Ballot`, representing the vote entity;
- `Builders` entities to strengthen the control over features in the building phase;
- `ElectionFactory` is the interface that hides the verbosity of builders;
- `ElectionManager` is the interface exposing method to manage the vote-casting process, including all verifications.

Codes In Figure 8 are summarized the main entities of the domain model concerning codes management. In particular:

- `OneTimeCode` represents the one time code entity;
- `CodesGeneratorStrategy` represents the strategy (i.e. the algorithm) used to generate the codes;
- `CodesRepository` is the interface managing the storage and retrieval of the codes. This interface is agnostic of the storage technology and will possibly be implemented in several ways, depending on the adopted one.
- `CodesManager` is the interface exposing methods to manage the creation, verification and (in)validation of codes.

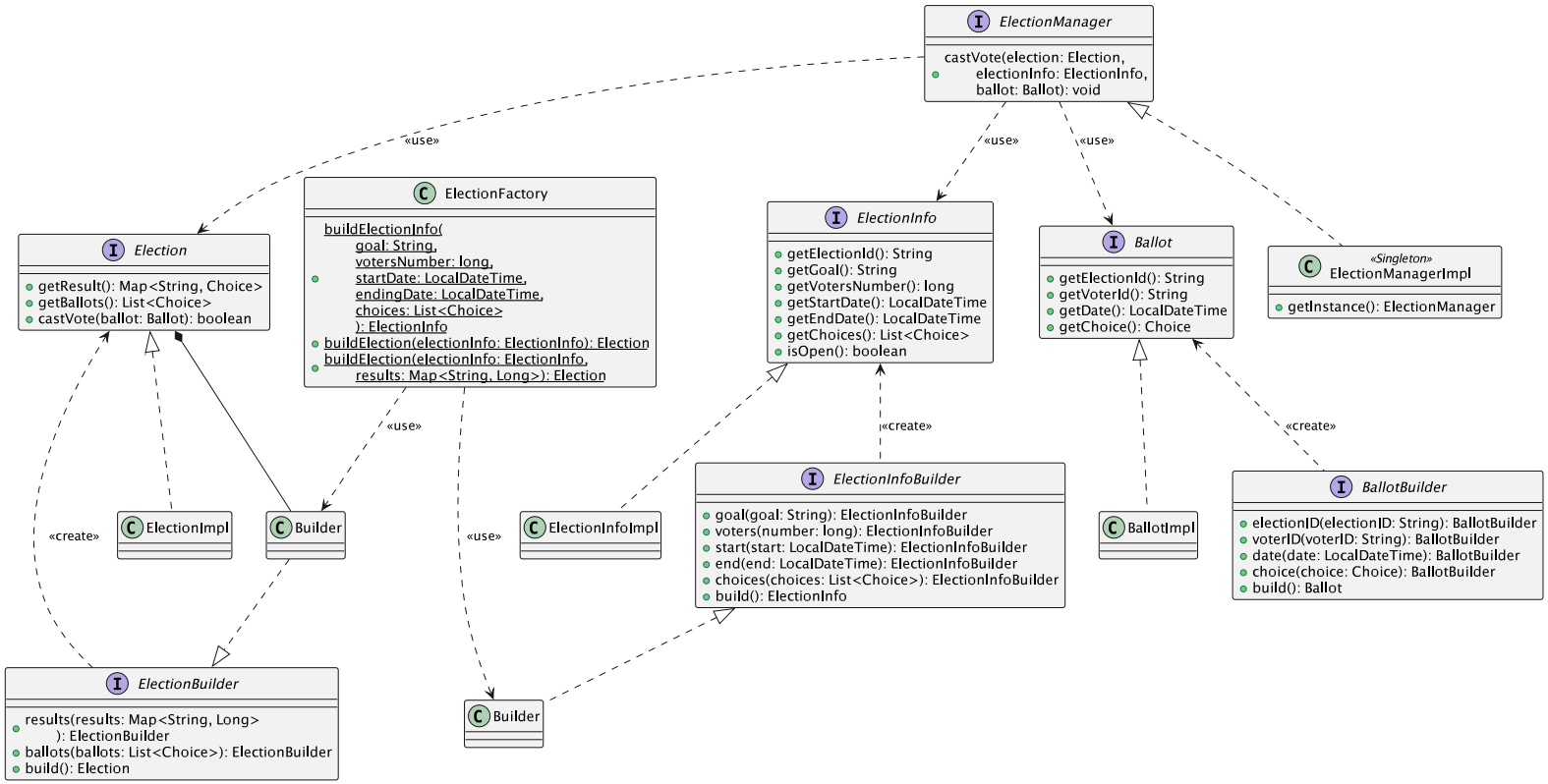


Figure 7: UML class diagram of the election's domain entities.

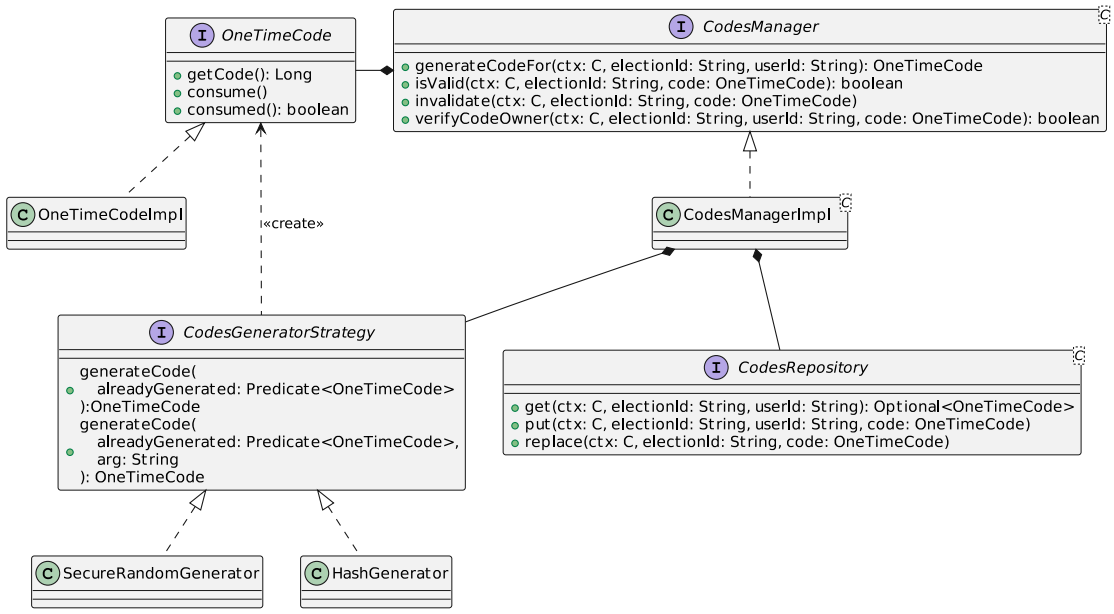


Figure 8: UML class diagram of the code's domain entities.

4.2.2 Interaction

In the following section are presented the most relevant (non-trivial) interactions grouped by use cases.

Use case: authentication to the system To interact with the system a user must request an authentication token from the server first (see Figure 9). To do this it must provide a valid username and password, then the server will check if the account exists and if the password is correct. If authentication is successful, the server will sign a pair of JWT tokens and send them to the user. These two will fill two distinct roles:

- **access token**: is needed to authenticate the user's requests to the system;
- **refresh token**: is needed to renew authentication without the user passing his credentials again.

Both must be stored securely once they have been transmitted. The access token has a short lifetime, while the refresh token has a longer one. When the access token expires, the user must request a new one using the refresh token. If the refresh token expires, the user must log in again.

Use case: casting a new vote for a given election To cast a new vote, the user first requests the generation of a new one-time code for a specific election. At the blockchain level, some checks are then performed in order to guarantee the election exists and a code has not already been generated for that user for the given election. Once the code

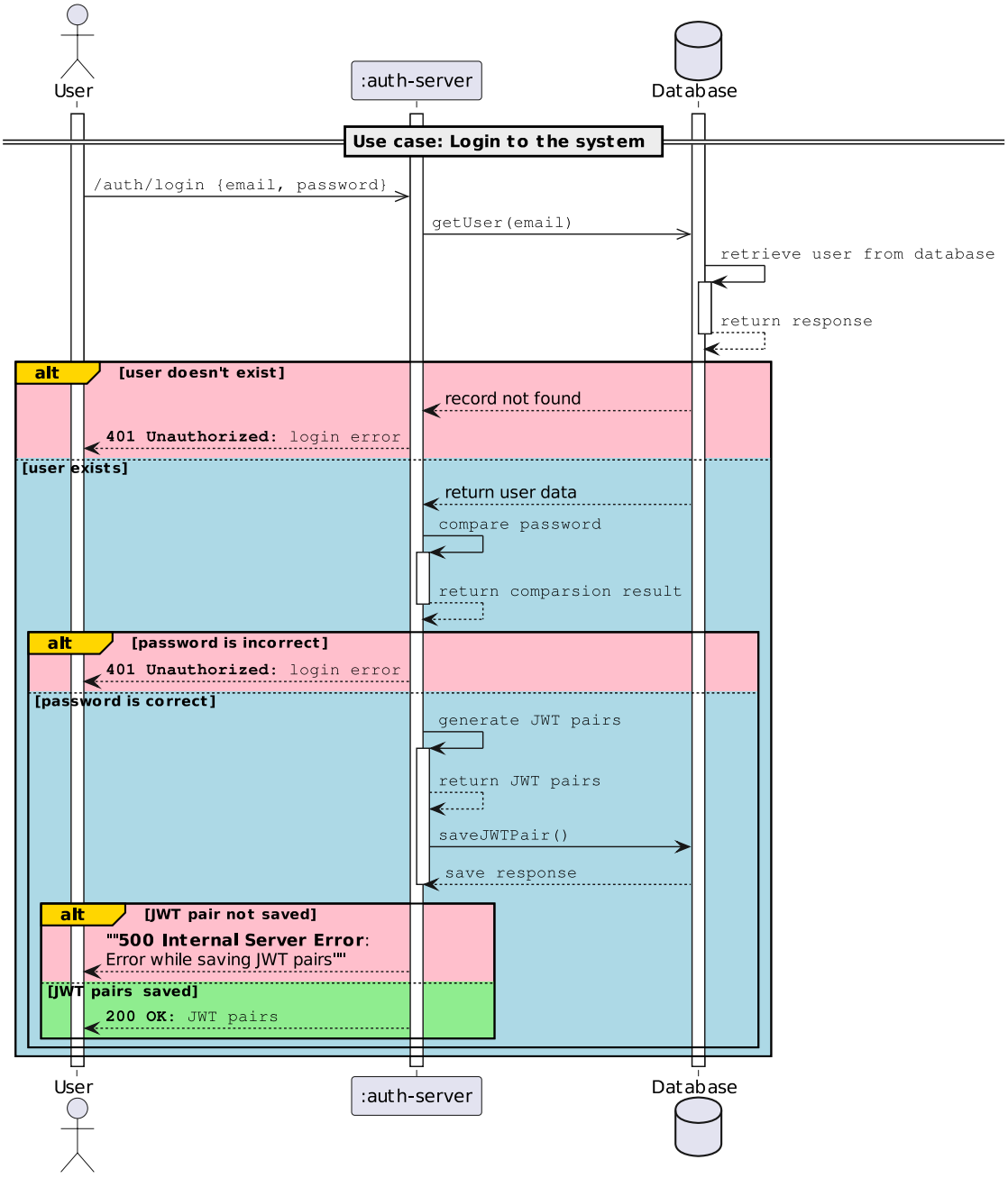


Figure 9: UML sequence diagram for the login use case.

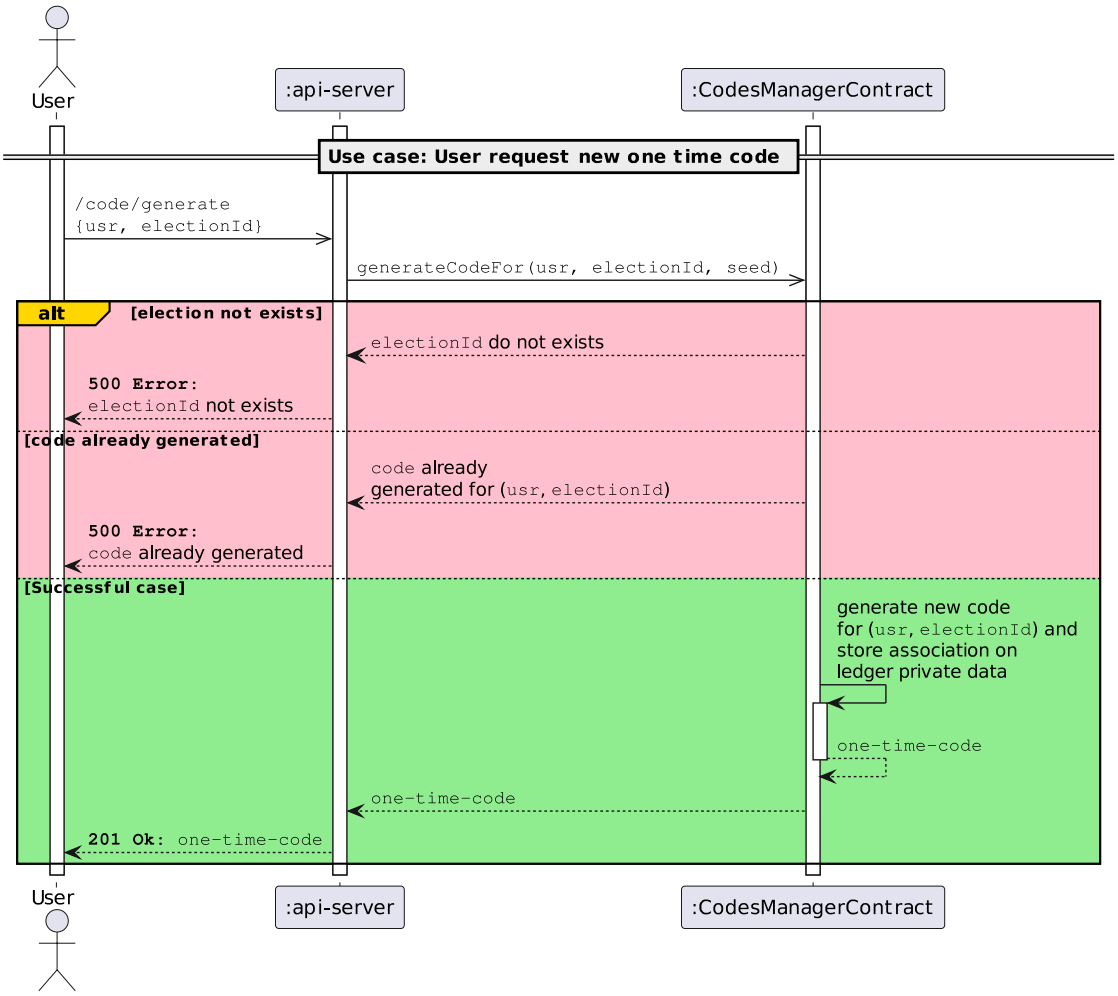


Figure 10: UML sequence diagram for the code's request use case.

has been generated, the API server is in charge of transmitting it to the user (using a trusted method) (see Figure 10). Upon receiving the code the user can cast their vote. Again, at the smart contract level, several checks are performed in order to ensure that the writing to the ledger of the newly cast vote is successful only if the election is open and the code is still valid, i.e. it is associated with the user who is requesting to vote and has not already been used. Finally, the code is invalidated (see Figure 11). The fact the code check and its invalidation are performed within the smart contracts of the blockchain ensures that it either succeeds and all peers agree on that or fails and the transaction is not recorded into the ledger. If these two actions were invoked from the API service a crash of the server in between a successful cast and code invalidation may allow the user to cast, at least theoretically, an infinite number of times, unless appropriately managed.

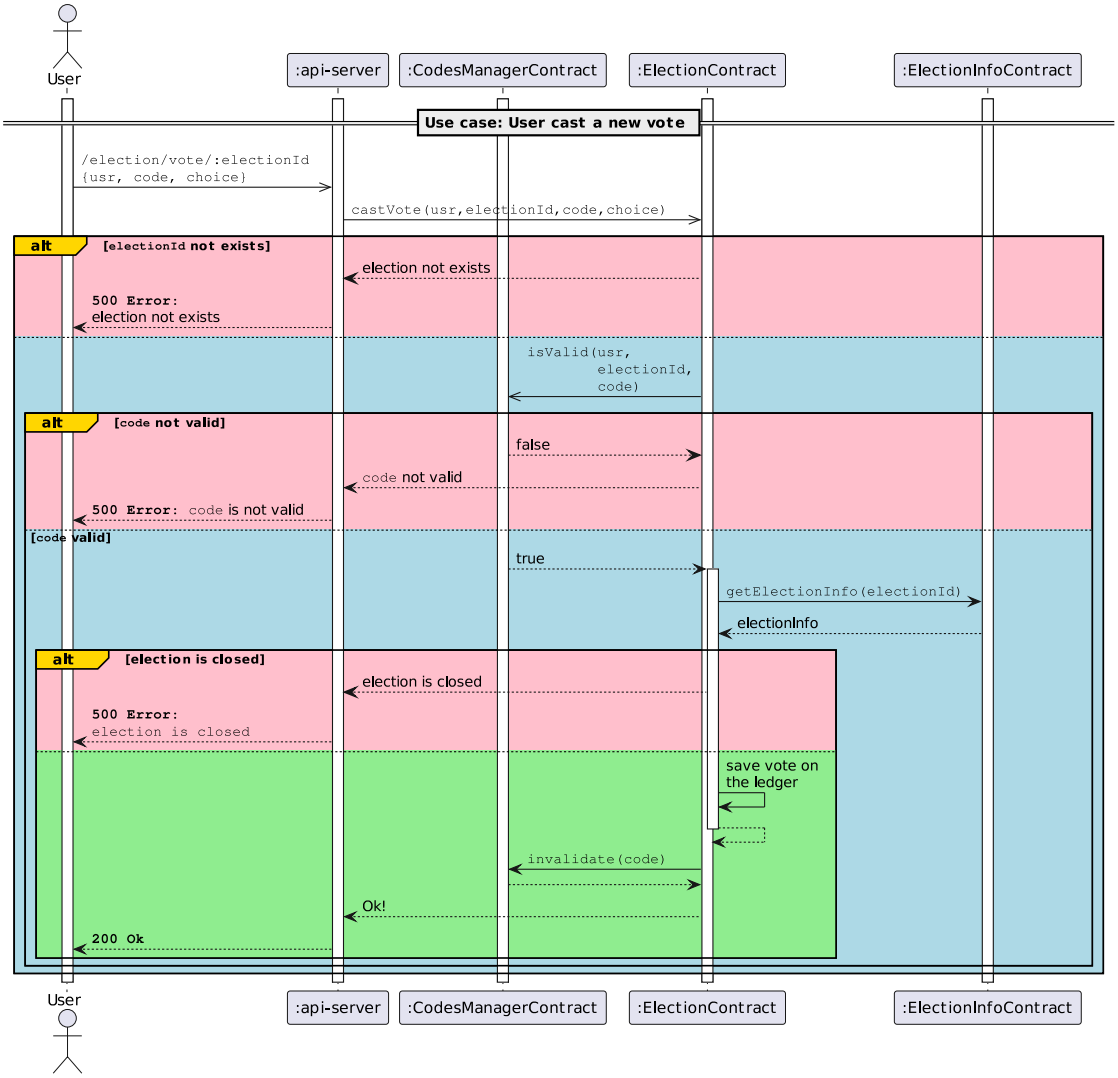


Figure 11: UML sequence diagram for the vote casting use case.

4.3 Architectural design

The architecture of the system is composed of three main components (see Figure 12):

- `:blockchain` component is in charge of the configuration and creation of the network's artifacts, as well as its deployment.
- `:api-service` module exposes the API to interact with the system.
- `:smart-contracts` module contains the implementation of the blockchain smart contracts.

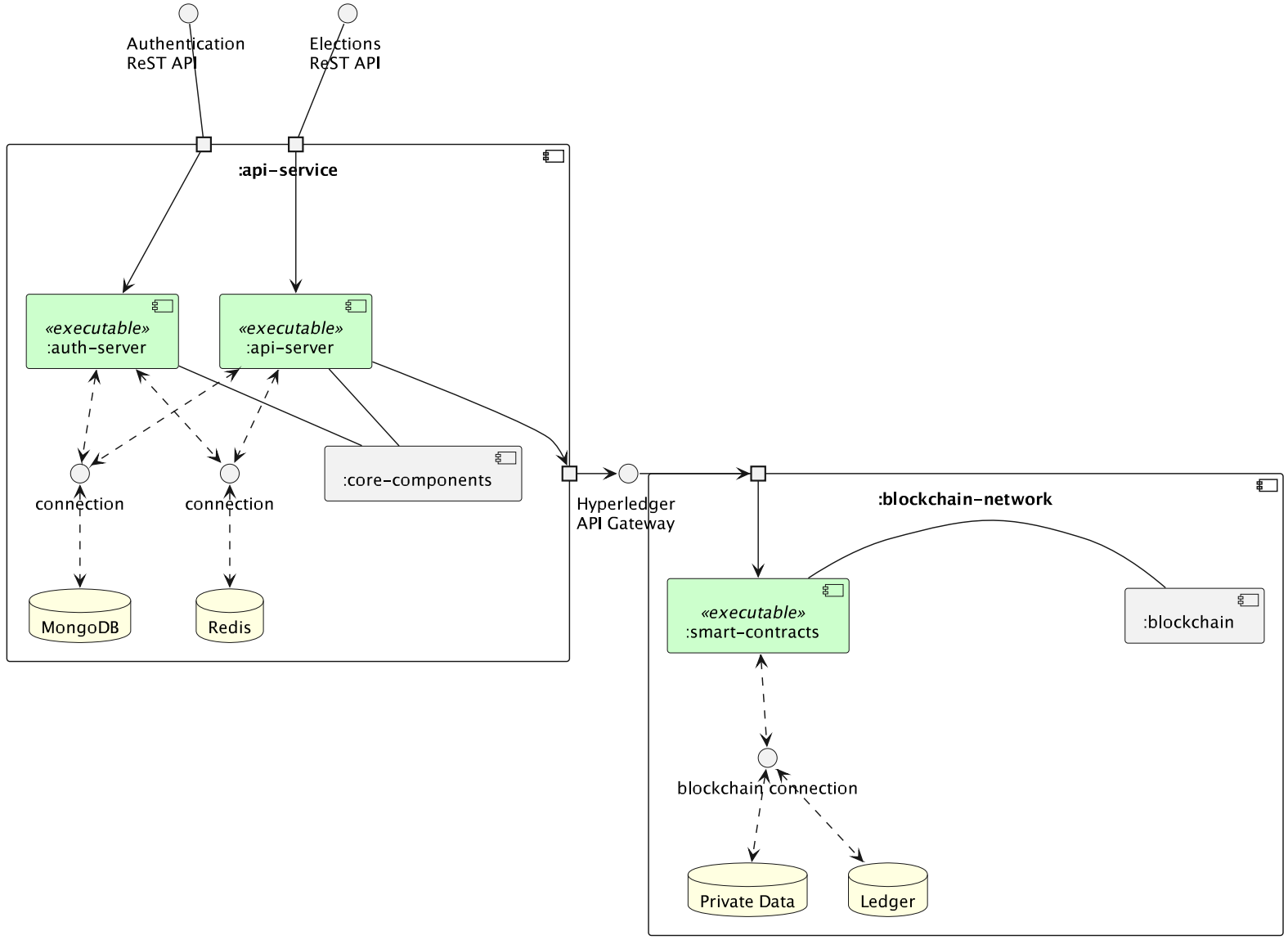


Figure 12: System architecture.

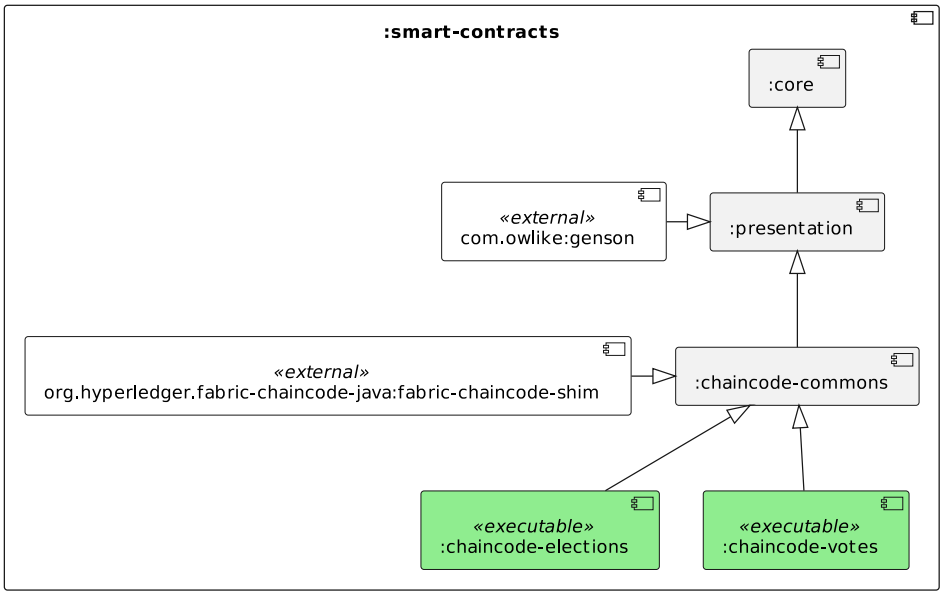


Figure 13: `:smart-contracts` component architecture.

Concerning the `:smart-contracts` component it consists of a JVM project with different sub-modules. It's structure is presented in Figure 13:

- the `:core` submodule captures the entities and interfaces of the domain model, which are technology-independent;
- a `:presentation` submodule which deals with de/serializations;
- a `:chaincode-commons` submodule which collects all the common utility classes for the development of the Hyperledger Fabric contracts;
- two submodules – `:chaincode-elections` and `:chaincode-votes` – which contains the chaincodes managing, respectively, the elections information and the votes, and encapsulate the logic of interaction with the network and its components using the technology-dependent API abstractions (in our case the Hyperledger Fabric chaincode shim library).

The internal architecture of this component decouples as much as possible the core business logic from external concerns, like technologies and used framework, making the system more maintainable and extensible: changing the technology with which to implement the blockchain in no way impacts the domain and its logic, which are assumed to remain stable over time.

The `:api-service` module exposes the core functionalities of the system through two submodules:

- `:api-server` used for handling the interaction with the blockchain network and the user's management;
- `:auth-server` used for authentication purposes.

We've separated the two functionalities to obtain a more modular and extensible system. Even if the authentication service stops working, the system will still be able to provide its functionality to users who are already authenticated. Both `:api-server` and `:auth-server` submodules are based on the Express.js framework and share some common functions (`:common`).

Other than the blockchain the system relies on two databases:

- A `MongoDB` service to store persistent information, i.e., user data and authentication tokens.
- A `Redis` database that is used for storing temporary data, i.e., the number of requests for a specific endpoint.

5 Implementation Details

The interesting implementation details of the project, divided by blockchain network, API and smart contracts, are presented below.

5.1 Blockchain Network

This section focuses on the most important network configurations, primarily focusing on the channel ones. The overall process of creation and deployment of the network and its component on top of Docker containers, as well as the CAs configurations and identities registration, is described in Section 7.3.

The channel artifacts are generated starting from a set of YAML configuration files, which are presented hereafter. All the configuration files and scripts concerning the channels can be found inside the `channels_config` folder of the `blockchain` module.

`configtx.yaml`

This configuration file is used to define and configure the network, including the organization's definitions, policies and other important parameters. In the following are presented the most relevant sections:

- **Organizations**: this section defines the different organizations that participate in the network. Each organization has a name, an ID, and specifies its cryptographic material (`MSPDir` field), as well as a set of ***signature policies***: each of them has a rule that specifies the set of organizations and identities whose signatures can satisfy the policy. In other words, they define the access control rules within an organization, governing who has permission to write, read, administer the channel and endorse (i.e. sign) a smart contract transaction. The syntax supports arbitrary combinations of `AND` and `OR`. The `org2` definition of this project's network is shown in Listing 1: for example, the *Admins* policy can only be satisfied by transactions submitted by an identity with an admin role, while only identities with a peer role can satisfy the *Endorsement* policy; all clients, peers and admin of both `org2` and `org3` can read, while only admin and client identities of `org2` are allowed to write.

Listing 1: `org2` definition.

```
Organizations:
  - &org2
    Name: org2MSP
    ID: org2MSP
    MSPDir: $(ARTIFACTS_DIR)/org2/msp
    # Organization policies canonical path:
    #   /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
    Policies:
        Readers:
            Type: Signature
            Rule: "OR('org2MSP.member', 'org3MSP.member')"
        Writers:
            Type: Signature
            Rule: "OR('org2MSP.admin', 'org2MSP.client')"
        Admins:
            Type: Signature
            Rule: "OR('org2MSP.admin')"
        Endorsement:
            Type: Signature
            Rule: "OR('org2MSP.peer')"
    AnchorPeers:
        - Host: peer1-org2
          Port: 9051
```

- **Application**: defines the policies that govern how peer organizations can interact with application channels. In particular, here we specify `ImplicitMeta` policies, which is a Hyperledger Fabric jargon to specify that they are compositions of the simpler *signature policies* defined in the `Organizations` section. All of them follow the following pattern: `<ANY|ALL|MAJORITY> <SubPolicyName>`. In our case, for channel 2, since we have chosen a `MAJORITY` application endorsement policy, both `org2` and `org3` must agree on the transaction response (see Figure 14): the criterion by which each organization approves is driven by its *signature* endorsement policy defined above; in this case is sufficient one peer per organization approval. Therefore, for every transaction concerning channel 2 we expect one peer of each organization to be contacted, and in order to be valid, both must agree on the outcome.

Listing 2: Application definition.

```
Application: &ApplicationDefaults
  Organizations:
    - *org1
    - *org2
  # Canonical policy path:
  #   /Channel/Application/<PolicyName>
  Policies:
    Readers:
      Type: ImplicitMeta
      Rule: "ANY Readers"
    Writers:
```

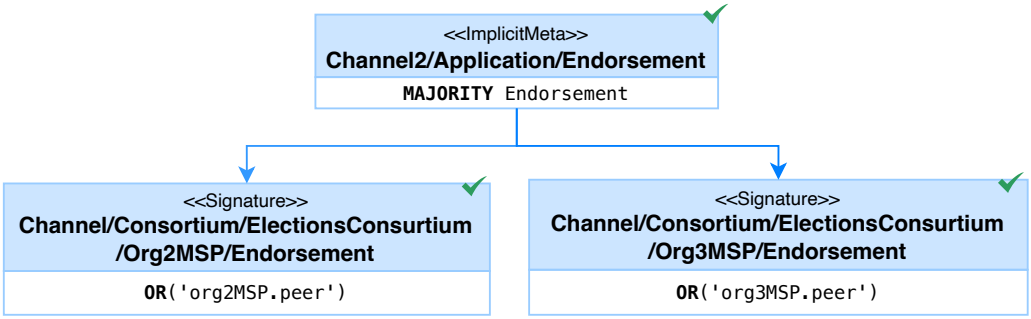


Figure 14: Policy configuration hierarchy.

```
      Type: ImplicitMeta
      Rule: "ANY Writers"
    Admins:
      Type: ImplicitMeta
      Rule: "MAJORITY Admins"
    # who needs to approve a chaincode definition
    # in the chaincode installation process
    LifecycleEndorsement:
      Type: ImplicitMeta
      Rule: "ANY Endorsement"
    Endorsement:
      Type: ImplicitMeta
      Rule: "MAJORITY Endorsement"
```

- **Orderer**: This section contains configuration details for the orderer nodes, such as their addresses, consensus type, the amount of time to wait before creating a batch, the batch size, and other relevant parameters. Moreover, are specified the policies that, similarly to the Application section, govern the ordering nodes. The chosen consensus is the `etcdraft` implementation.

Listing 3: Orderer definition.

```
Orderer: &OrdererDefaults
  OrdererType: etcdraft
  EtcdRaft:
    Consenters:
    - Host: orderer1-org0
      Port: 7050
      ClientTLSCert: $(ARTIFACTS_DIR)/org0/orderer1/tls-msp/signcerts
          /cert.pem
      ServerTLSCert: $(ARTIFACTS_DIR)/org0/orderer1/tls-msp/signcerts
          /cert.pem
    - ...
  BatchTimeout: 2s
  BatchSize:
    ...
  Organizations:
    - *org0
```

```
  # Canonical policy path:
  #   /Channel/Orderer/<PolicyName>
  Policies:
    Readers:
      ...
    Writers:
      ...
    Admins:
      ...
    # BlockValidation specifies what signatures must be included
    # in the block from the orderer for the peer to validate it.
    BlockValidation:
      ...
```

- **Channel**: this section defines the policies that govern the highest level of the channel configuration. In an application channel, these policies govern the hashing algorithm, the data hashing structure used to create new blocks, and the channel capability level. Default policies are provided by Hyperledger Fabric which, in most cases, there is no need to change.

Listing 4: Channels definition.

```
Channel: &ChannelDefaults
  # Canonical policy path:
  #   /Channel/<PolicyName>
  Policies:
      # Who may invoke the 'Deliver' API
      Readers:
          Type: ImplicitMeta
          Rule: "ANY Readers"
      # Who may invoke the 'Broadcast' API
      Writers:
          Type: ImplicitMeta
          Rule: "ANY Writers"
      # By default, who may modify elements at this config level
      Admins:
          Type: ImplicitMeta
          Rule: "MAJORITY Admins"
  Capabilities:
      <<: *ChannelCapabilities
```

- **Profiles**: The `configtxgen` tool reads this section to build a channel configuration. Each profile uses YAML syntax to gather data from other sections of the file.

`core.yaml` and `orderer.yaml`

These files are used to configure settings for the core components of, respectively, a peer node and an orderer node.

Among the plethora of options that can be configured the focus is on the following:

- peer connectivity information, like identifier, network identifier, the address at local network interface the peer will listen on and gossip configurations;

- `tls` is enabled;
- ledger world state database has been set to CouchDB for the reasons already discussed in Section 2.1.

Note that, for this project, a single `core.yaml` and `orderer.yaml` files have been configured with common configurations for all the peers and orderers. The options intrinsically related to peer and orderers instances, like the address, will be overridden during the creation of the network with environment variables. This guarantees to have a single template for all the peers with static configuration while configuring dynamic ones at channel creation time with scripts.

5.2 Smart Contracts

Hyperledger Fabric smart contracts are implemented through classes annotated with `@Contract` implementing the chaincode-shim `ContractInterface`, while transactions are simple methods labelled with `@Transaction(intent = ...)` where the `intent` specifies the semantics for the transaction:

- `Transaction.TYPE.SUBMIT` indicates that this function is intended to be called with the *submit* semantics, i.e. the transaction will update the ledger and modify the state of the blockchain, producing a lasting impact on the network's ledger;
- `Transaction.TYPE.EVALUATE` indicates that this is intended to be called with the *evaluate* semantics, i.e. the transaction is read-only and should not make any updates to the ledger. It is primarily used for querying the blockchain to retrieve information or perform calculations without affecting the ledger's state.

The general structure of a smart contract is presented in Listing 5.

Listing 5: General structure of a Hyperledger Fabric smart contract

```
import org.hyperledger.fabric.contract.*;
import org.hyperledger.fabric.contract.annotation.*;

@Contract(
    name = "Sample Smart Contract",
    info = @Info(
        title = "A simple contract",
        description = "An implementation of a smart contract in Java"
    ),
    transactionSerializer = "it.unibo.ds.MyTransactionSerializer"
)
public class HFSmartContract implements ContractInterface {

    @Override
    public void beforeTransaction(Context ctx) {
        // Custom logic to be invoked before each transaction
        ContractInterface.super.beforeTransaction(ctx);
    }

    @Transaction(intent = Transaction.TYPE.SUBMIT)
```

```
    public void myTransaction(Context ctx, String arg1, int arg2) {
        // Transaction logic
    }

    @Override
    public void afterTransaction(Context ctx, Object result) {
        // Custom logic to be invoked after each transaction
        ContractInterface.super.afterTransaction(ctx, result);
    }
}
```

A few things to notice:

- inside the `@Contract` annotation is possible to specify a custom `SerializerInterface` which will be used to (un)marshall transaction inputs;
- every transaction receives a transaction context, which Hyperledger Fabric automatically provides when a smart contract is invoked. This context includes both system and user elements, enabling smart contract developers to interact with the ledger and store or retrieve user-defined data across consecutive transaction calls. An overview of the main functionalities offered by the chaincode-shim library is described in the UML class diagram shown in Figure 15.

Other important points to highlight are how the one-time codes are generated, where they are saved and the mechanism used for hiding the transaction inputs.

As stated in Section 3.1 for the generation of codes a deterministic algorithm has been used, hashing the static information of the election with a randomly generated seed passed to the blockchain by the API server. We assume the API server is a trustworthy component that is impossible to tamper with.

Codes are then stored inside **Private Data** collections. This ensures that only organization 2, the one in charge of the creation of codes, has a copy and can access the private collection where they are stored.

From an implementation point of view accessing a private data collection is almost the same as accessing the world state: this implies that private data collections are merely an implementation detail that is completely transparent to the client application.

Listing 6: Examples of how to retrieve and store data on a private collection

```
/* Note: a Context object is expected to be available in the scope of
 * this code (it is passed as an argument to the transaction functions).
 */
final String CODES_COLLECTION = "CodesCollection";

// example of how to persist data in a private collection
context.getStub().getPrivateData(
    CODES_COLLECTION,
    new CompositeKey(electionId, userId).toString()
);

// example of how to retrieve data from a private collection
```

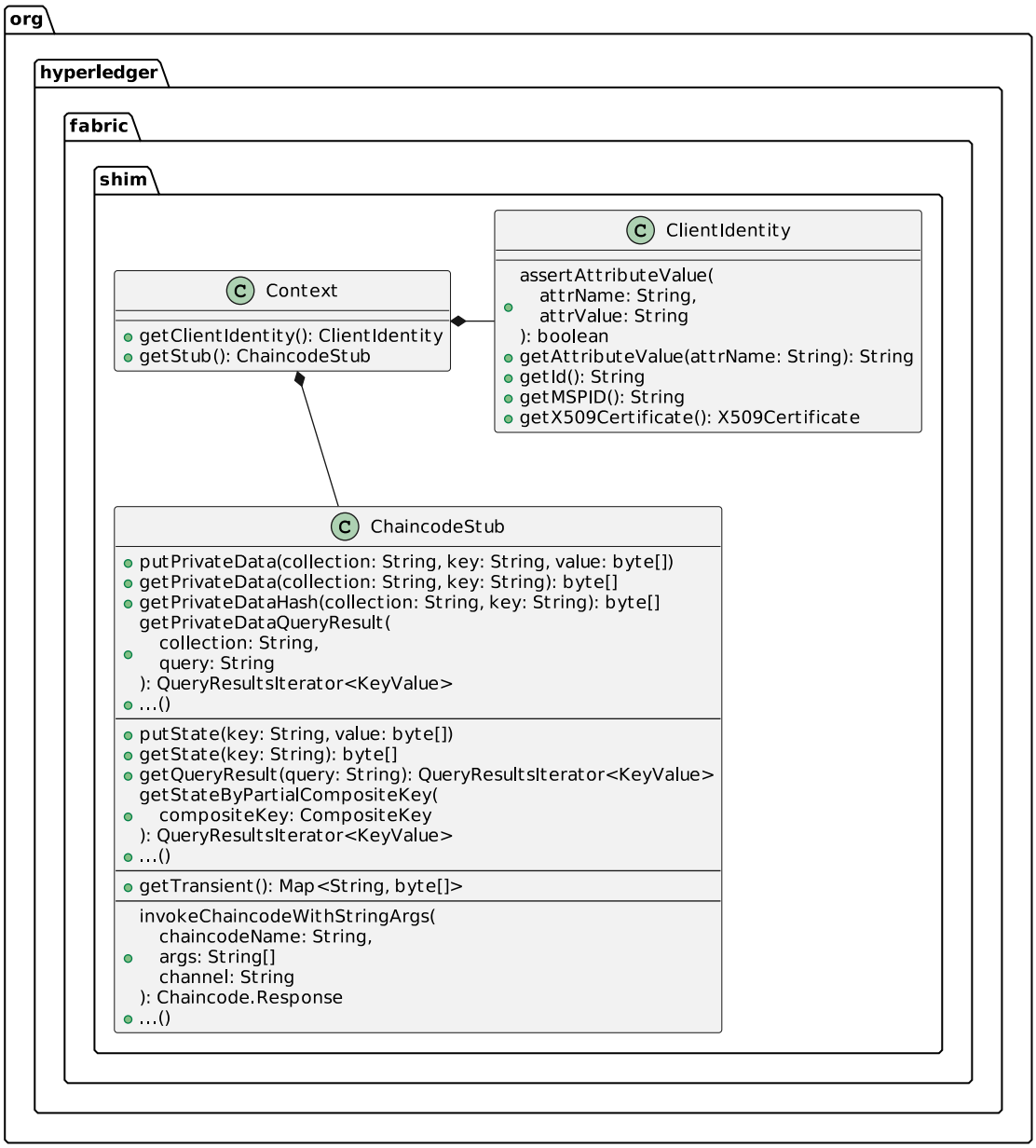


Figure 15: Overview of the main functions offered by the chaincode-shim API to interact with the ledger.

```
context.getStub().putPrivateData(
    CODES_COLLECTION,
    new CompositeKey(electionId, userId).toString(),
    genson.serialize(new OneTimeCodeAsset(electionId, userId, code))
);
```

Private data collections can be configurable in terms of endorsement policy, time to live (in terms of the number of blocks), access control and information dissemination through a JSON configuration file (see `smart-contracts/chaincode-votes/src/main/resources/collections-config.json`). In particular:

- `policy` defines the organization peers allowed to persist the collection data;
- `requiredPeerCount` defines the minimum number of peers that must receive and store the private data in order for an endorsement to be considered valid;
- `maxPeerCount` the number of other peers that the current endorsing peer will attempt to distribute the data to.
- `blockToLive` represents how long the data should live on the private database in terms of blocks. To keep indefinitely set to 0;
- `memberOnlyRead` is a boolean value indicating that only clients belonging to one of the collection member organizations have read access to private data;
- `memberOnlyWrite` a boolean value indicating that only clients belonging to one of the collection member organizations have write access to private data;
- `endorsementPolicy` defines the endorsement policy that needs to be met in order to write to the private data collection.

Listing 7: Private code collection configuration

```
[
  {
    "name": "CodesCollection",
    "policy": "OR('org2MSP.member', 'org3MSP.member')",
    "requiredPeerCount": 1,
    "maxPeerCount": 1,
    "blockToLive": 0,
    "memberOnlyRead": true,
    "memberOnlyWrite": true,
    "endorsementPolicy": {
      "signaturePolicy": "AND('org2MSP.peer', 'org3MSP.peer')"
    }
  }
]
```

Until now we have discussed how to store data in private collections but we have not yet considered a potential vulnerability: Hyperledger Fabric records the submitter's digitally signed transaction inputs, as well as the digitally signed transaction responses. Therefore, if not protected in some way, the user and related one-time codes are available for all to see, as they are recorded in the transaction distributed to the network. In

order to avoid this issue Hyperledger Fabric offers a mechanism, called **transient data**, whereby the transaction inputs are not recorded anymore. When a client initiates a transaction, it can attach transient data to the transaction proposal: the chaincode-shim API, as well as the Gateway one, exposes methods allowing the retrieving and packing of data inside transient maps (Figure 15). This data is then endorsed by the required peers and included in the transaction payload during the ordering process. However, unlike the actual transaction data, transient data is not recorded on the blockchain ledger and is not visible to other nodes in the network.

Therefore, transient data have been used to hide the associations between code and user, as well as, expressed choice and code.

Due to the blockchain architecture choice of decoupling election information, the application requires a way to access static information in the chaincode that manages the votes. To achieve this, an example of cross-chaincode invocation is provided in Listing 8.

Listing 8: General structure of a Hyperledger Fabric invocation of a method located in a different chaincode

```
Response response = ctx.getStub().invokeChaincodeWithStringArgs(
    CHAINCODE_NAME,
    List.of("ContractName:methodName", arg1, arg2, ...),
    CHANNEL_NAME
);
if (response.getStatus().equals(Response.Status.SUCCESS)) {
    try {
        Response<MethodReturnType> payload = genson.deserialize(
            response.getStringPayload(),
            new GenericType<>() { }
        );
        ...
    } catch (JsonBindingException | NullPointerException e) {
        throw new ChaincodeException(
            e.getMessage(),
            Error.CROSS_INVOCATION_ERROR.toString()
        );
    }
} else {
    var errorMessage = "Cross invocation failed.";
    throw new ChaincodeException(
        errorMessage,
        Error.CROSS_INVOCATION_FAILED.toString()
    );
}
```

Documentation Here are listed the links where you can find the JavaDoc of the `:smart-contracts` module:

- `:chaincode-elections` API
- `:chaincode-votes` API

(`:core` and `:presentation` submodules' APIs are here reported for completeness although not necessary for the ReST API service client.)

5.3 Gateway

To interact with the blockchain network, starting from version 2.4, the Hyperledger Fabric peers expose a new gRPC service, called `Gateway`, which API allows to submit transactions to the network and retrieve data from the ledger. To connect to the blockchain with the Gateway service we need to establish a new gRPC session, select a channel and a smart contract we want to invoke. The `Communicator` class was developed to simplify the overall configuration process (Figure 16).

`CommunicatorInterface` define three methods:

- `createGrpcClients` establishes a new gRPC session with the peer;
- `createIdentity` creates an `Identity` object that represents the identity that will interact with the blockchain; for doing this it will use the certificates created by the CA during the creation of the network;
- `createSigner` creates a `Signer` object representing the signer of the transactions.

The Communicator class requires several arguments for the instantiation, and this could represent a usage problem, for this purpose the `CommunicatorBuilder` and `CommunicatorFactory` were developed. The first one is a builder that allows the construction of a Communicator object in a more readable way, while the latter is a factory class that creates instances of Communicator that will connect to a target peer that will act as a gateway endpoint. Since the functions exposed by the endpoints of the API server need to use a gRPC session every time they are invoked, to make the creation process less burdensome we delegate the instantiation and management of the gRPC clients to the class `GrpcClientPool`.

Listing 9: An example of an API method that communicates with the blockchain network

```
export async function getAllElection(req: Request, res: Response, next:
    NextFunction) {
    if(!ac.can(res.locals.user.role).readAny('election').granted) {
        next(new UnauthorizedError("Can't access to the resource"));
    }
    try {
        const gatewayOrg2: Gateway = await GrpcClientPool
            .getInstance()
            .getClientForPeer(Org2Peer.PEER1);
        const network: Network = gatewayOrg2.getNetwork(channelName);
```

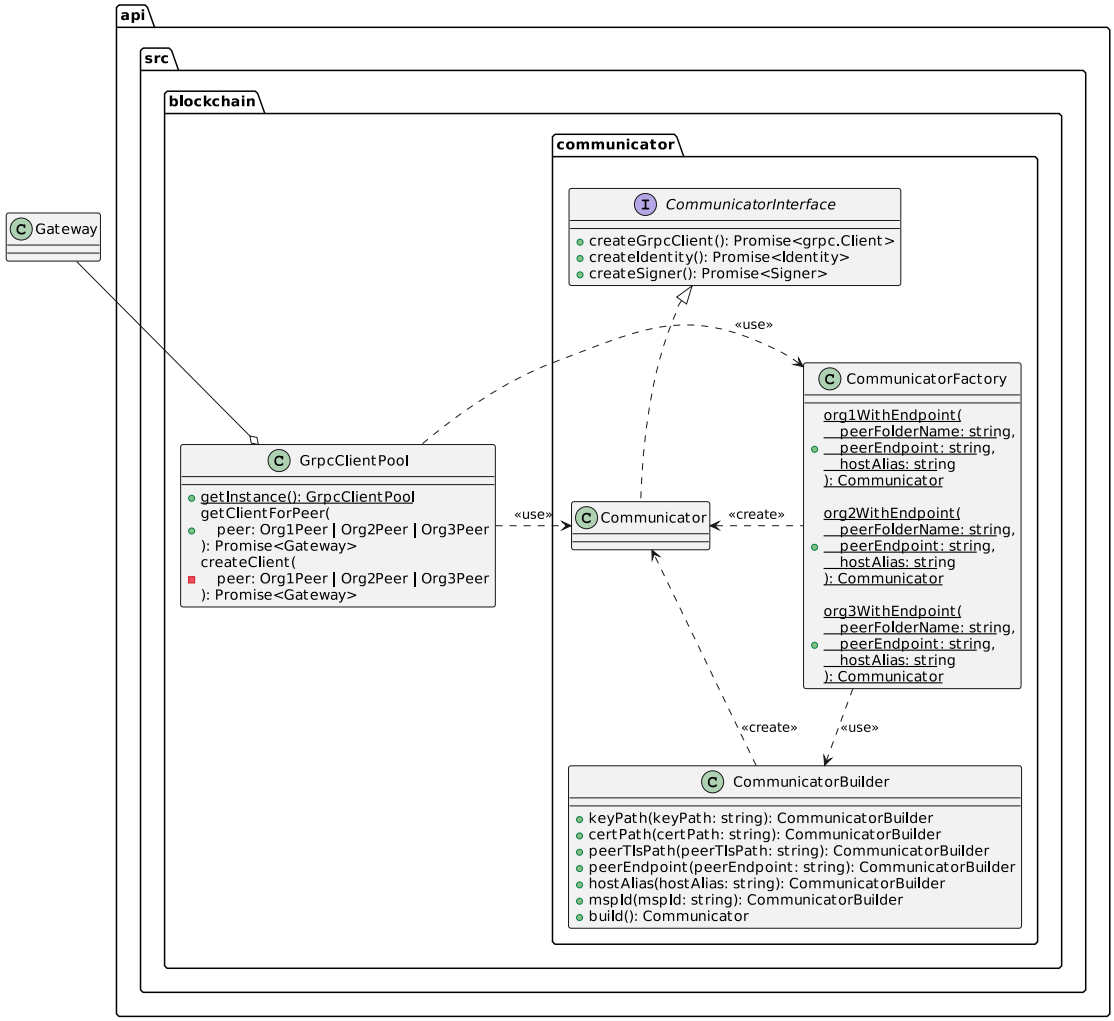


Figure 16: The communicator API component that allows the connection to the Gateway service configuration.

```
        const contract: Contract = network.getContract(contractName);
        const submission: Uint8Array = await contract
            .evaluate('ElectionContract:getAllElection', {
                arguments: []
            }
        );
        const result = utf8Decoder.decode(submission);
        res.locals.code = StatusCodes.OK;
        res.locals.data = JSON.parse(result).result;
    } catch (error) {
        return next(transformHyperledgerError(error));
    }
    return next;
}
```

The first time that we request a client for a specific endpoint the GrpcClientPool will create a new gRPC session and will store it in a map, associating it with the endpoint. The next time that we request a client for the same endpoint, the GrpcClientPool will return the client that was created previously. This will allow us to reuse the same gRPC session for multiple requests, without having to create a new one.

5.4 Authentication

Authentication within the system is implemented by the use of `JWT tokens`. JWT [4] is an open standard that allows the secure transmission of information between two parties as a JSON object. The information passed is digitally signed using the RS256 algorithm so it can be trusted and verified by an entity that requests it. Both the sign and verify functions for an access token are distributed to the `API` and `AUTH` server inside the `:common` package.

As mentioned in Section 4.2.2, when a user tries to log in, the `AUTH` server will sign an access and a refresh token. The first one has a validity of 15 minutes while the latter one of 30, both will be securely signed by a private key and then sent back to the client.

Listing 10: The login method in the auth server

```
export async function login(req: Request, res: Response, next:
    NextFunction) {
    const email = req.body.email;
    const password = req.body.password;
    try {
        const user = await User.findOne({email: email});
        //...
        user.comparePassword(password, async function(error, isMatch) {
            // If password is ok, create token pair and send it to client
            const tokens = await Jwt.createTokenPair(user);
            res.locals.code = StatusCodes.OK;
            res.locals.data = {
                accessToken: tokens.accessToken,
                refreshToken: tokens.refreshToken
            };
```

```
            //...
        });
    } catch(error) {
        //...
    }
}
```

The client is responsible for securely storing the tokens and for sending them to the request that requires authentication. Before accessing the routes, `API` server verifies the validity of the access token and extract the information that it needs from it. This operation is performed by a middleware that is executed before the request is processed by the route handler. It verifies the validity of the token and extracts the information that it needs from it. If the token is valid, the middleware registers the user information and sends it to the function that handles the request. If the token is not valid, the middleware sends back an error response.

Listing 11: The middleware that handles the authentication

```
export async function authenticationHandler (req: Request, res: Response,
     next: NextFunction) {
    const authorizationHeader = req.headers["authorization"];
    if(authorizationHeader == undefined) {
        return next(new BadRequestError("Authorization header not found
            in the request"));
    }
    const accessToken = authorizationHeader.split(" ")[1];
    try {
        const jwtRecord = await Jwt.findOne({accessToken: accessToken});
        if(jwtRecord == null) {
            return next(
                new NotFoundError("Submitted jwt token doesn't exists")
            );
        }
        const validationResponse:any = await jwtRecord.
            validateAccessToken();
        const user = await User.findOne(
            {email: validationResponse.sub.email}
        );
        if(user == null) {
            return next(new NotFoundError("User not found"));
        }
        res.locals.user= user;
    } catch(error) {
        return next(error);
    }
    next();
}
```

The `Jwt` model handles the verification and persistence of tokens inside the `MongoDB` database. It uses the namesake utility class, located within the `utils.jwt` package which is responsible for managing the construction of the various parts of the token.

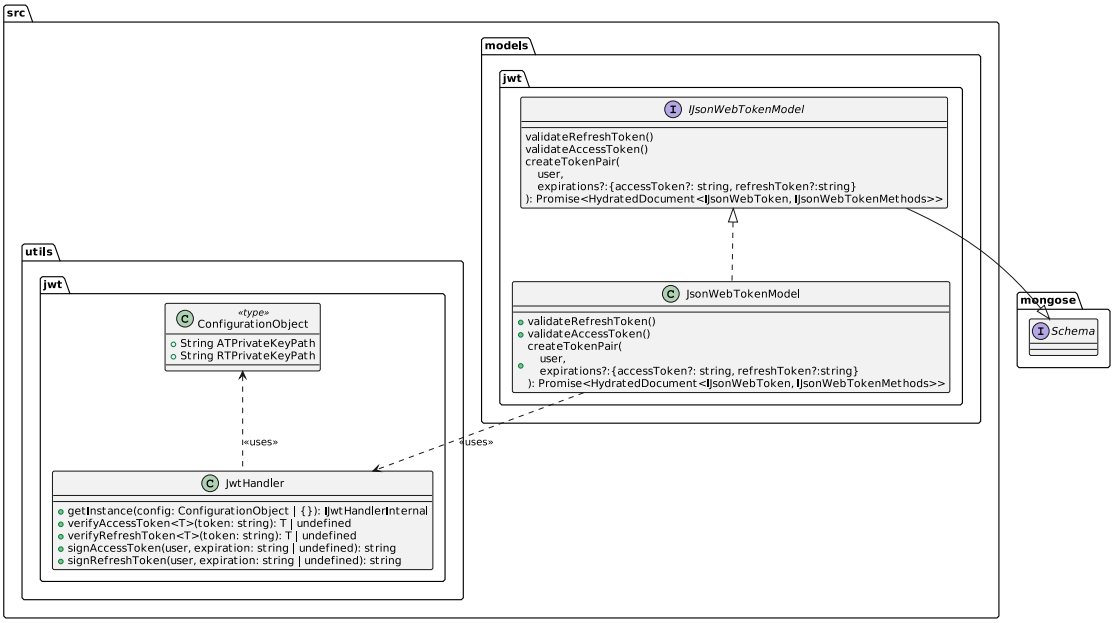


Figure 17: The overall API for the management of the JWT tokens

Warning Once the network is deployed, the script copies a pair of public & private keys in `api`, `auth` and `common` modules. While this operation simplifies the deployment process, and its testing, we're aware that in a real scenario, this would be a security issue and the private key should be stored and distributed more securely way.

5.5 Rate limiter

We've implemented a custom rate-limiter module inside the API layer, which is responsible for limiting the number of requests that a client can send to a specific endpoint. This is done by using a `Redis` database that stores the number of requests that a client has sent to a specific endpoint. The rate limiter is implemented as a middleware that is executed before the request is processed by the route handler. It checks if the client has exceeded the maximum number of requests allowed, if so it sends back an error response to the client, otherwise, it increments the number of requests and sends the request to the route handler.

The way it works is as follows: each controller specifies a set of rules for each endpoint that can vary from the type of request.

Listing 12: The definitions of the rules for the rate-limiter

```
1 const API_LIMITER_RULES: ApiLimiterEntry = {
2     "/endpoint_name_1": {
```

```
        "POST": {
            time: 18,
            limit: 50
        },
        "GET": {
            time: 14,
            limit: 60
        }
        //...
    },
    "/endpoint_name_2": {
        "POST": {
            time: 15,
            limit: 30,
        }
        //...
    },

    "/endpoint_name_3": {
        "PUT": {
            time: 20,
            limit: 100
        },
        "DELETE":  {
            time: 20,
            limit: 100
        },
        //...
    }
}
```

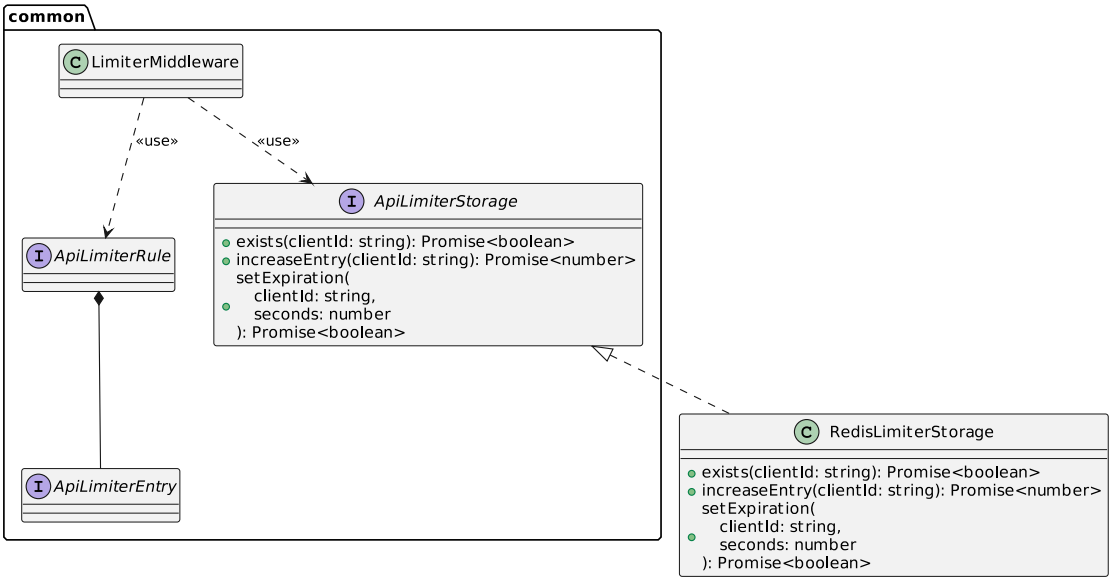
In Listing 12 we specified the rules for three endpoints, each of them specify for the `verb` type two parameters:

- `time`: The time window in which the requests are counted.
- `limit`: The maximum number of requests that can be sent in the specified time window.

So, for example, `endpoint_name_1` can receive a maximum of 50 requests in 18 seconds if the request type is `POST` and 60 requests in 14 seconds if the request type is `GET`. The set of rules is then passed to the RateLimiter middleware which retrieves the IP address of the client and the endpoint that is being requested. Route checking is done on data saved on a `Redis` database. To make this solution more modular and extensible a Storage (`ApiLimiterStorage`) interface was devised that defines the operations to be performed on the data.

Listing 13: The limiter middleware

```
export function apiLimiter(rules: ApiLimiterEntry, storage:
    ApiLimiterStorage) {
    return async function(req: Request, res: Response, next: NextFunction
        ) {
```




```
        const endpoint = getBaseEndpoint(req.path);
        const method = req.method;
        const clientId = `${method}/${endpoint}/${req.ip}`;

        if(!rules[endpoint] || !rules[endpoint][method]) {
            return next();
        }

        const currentEntries = await storage.increaseEntry(clientId);
        if(currentEntries > rules[endpoint][req.method].limit) {
            return next(new TooManyRequests("Api Limit Reached"));
        }

        await storage.setExpiration(clientId, rules[endpoint][method].
            time);
        return next();
    }
}
```

Documentation Here are listed the links where you can find the Open API documentation of both `:api-server` and `auth-server` modules:

- `:api-server` API
- `:auth-server` API

6 Self-assessment / Validation

Tests have been conducted on different levels: domain model, smart contracts and API interactions.

Concerning the domain model of the smart contract unit test has been used (using JUnit framework [8]). They can be found in `:core` module.

To test smart contracts, since their functioning requires the network to be up and running, and to effectively bring up the network requires a non-negligible time which would have made the testing environment viscous, Mockito framework [10] has been adopted to mock the network-dependent parts. Indeed, Mockito allows the creation of mock objects, which essentially are placeholders for real objects that simulate their behavior by defining the expected outcome provided by specified methods, and spy objects which allow a partial mock of some method of an existing object, retaining its original behavior while customizing certain aspects. An example of usage for the

Listing 14: An example of Mockito behavior definition

```
Context contextMock = mock(Context.class);
ChaincodeStub stubMock = mock(ChaincodeStub.class);
String MY_ELECTION_ID = "election_example"
ElectionInfo MyElectionInfo = ... // ElectionInfo creation

when(contextMock.getStub()).thenReturn(stubMock);
when(contextMock.getStub().invokeChaincodeWithStringArgs(
    CHAINCODE_INFO_NAME,
    List.of("ElectionInfoContract:readElectionInfo", MY_ELECTION_ID),
    CHANNEL_INFO_NAME
)).thenReturn(
    new Chaincode.Response(
        Chaincode.Response.Status.SUCCESS,
        "A payload example",
        genson.serialize(new Response<>(MyElectionInfo)).getBytes(UTF_8)
    )
);
```

Listing 15: An example of Mockito Spy behavior definition

```
Context contextMockSpy = spy(new Context());
ChaincodeStub stub = spy(new ChaincodeStub());
String MY_ELECTION_ID = "election_example"
ElectionInfo MyElectionInfo = ... // ElectionInfo creation

doReturn(stub).when(contextMockSpy).getStub();
doReturn(new Chaincode.Response(
    Chaincode.Response.Status.SUCCESS,
    "A payload example",
    genson.serialize(new Response<>(MyElectionInfo)).getBytes(UTF_8))
).when(contextMockSpy).getStub().invokeChaincodeWithStringArgs(
    CHAINCODE_INFO_NAME,
    List.of("ElectionInfoContract:readElectionInfo", MY_ELECTION_ID),
    CHANNEL_INFO_NAME
```

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	89.8% (44/49)	82% (182/222)	83.5% (664/795)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
it.unibo.ds.chainvote	55.6% (5/9)	55.6% (10/18)	60.3% (35/58)
it.unibo.ds.chainvote.asset	100% (1/1)	80% (4/5)	87.5% (7/8)
it.unibo.ds.chainvote.codes	100% (9/9)	74.4% (29/39)	82.6% (57/69)
it.unibo.ds.chainvote.contract	88.9% (8/9)	90.9% (30/33)	82.4% (140/170)
it.unibo.ds.chainvote.converters	100% (5/5)	100% (15/15)	93.1% (188/202)
it.unibo.ds.chainvote.elections	100% (6/6)	93.8% (45/48)	90.7% (107/118)
it.unibo.ds.chainvote.facade	100% (2/2)	35.7% (5/14)	63.3% (19/30)
it.unibo.ds.chainvote.facade.converter	100% (1/1)	50% (1/2)	5.6% (1/18)
it.unibo.ds.chainvote.factory	100% (1/1)	92.3% (12/13)	98% (50/51)
it.unibo.ds.chainvote.manager	100% (1/1)	100% (6/6)	92.9% (13/14)
it.unibo.ds.chainvote.utils	100% (5/5)	86.2% (25/29)	82.5% (47/57)

generated on 2023-11-30 21:05

Figure 18: Coverage of the `:smart-contracts` module.

```
15 );
```

Some testing metrics regarding `smart-contracts module`: a total of 86 tests were developed, including `core`, `presentation` and `chaincode*` submodules, and the test coverage is overall higher than 80% (the detail is shown in Figure 18).

Tests To run the smart contracts tests, simply: `./gradlew test` inside the `smart-contracts` folder of the project.
Reports are available in each `build/reports/tests` submodule directory.

Concerning distributed quality attributes of the blockchain component, such as fault tolerance, we leverage the Hyperledger Fabric framework which ensures that, for example, if a peer is down, the transactions are redirected to the other peers of the organization. Manual tests have been conducted to stress the network in corner cases (like the one cited above) operating on the peers' containers.

6.1 API Testing

The principal goal of the API testing is to verify that the server correctly handles the requests that it receives from the client. To achieve this we used in combination the `jest`

testing library along with `supertest`. The first one is a JavaScript testing framework that allows the creation of unit tests for JavaScript code, while the latter is a library that allows the creation of HTTP requests to a server

Listing 16: An example of an API test

```
beforeAll(async () => {
    app = ExpressConfig();
});

beforeEach(async () => {
    /**
     * INITIALIZATION CODE
     */
});

afterEach(async () => {
    /**
     * CLEANUP CODE
     */
});

describe("POST /code/", () => {

    test("Can check if a code is valid", async () => {
        const electionId = await createElection(app, adminJwtToken.
            accessToken);
        const code = await createCodeForElection(app, electionId,
            userJwtToken.accessToken);

        console.log(code, user.id, electionId);
        const response = await request(app).post("/code/check")
            .send({code: code, electionId: electionId })
            .set("Authorization", `Bearer ${userJwtToken.accessToken}`)
            .set("Accept", "application/json")
            .expect("Content-Type", "application/json; charset=utf-8")
            .expect(StatusCodes.OK);

        expect(response.body.success).toBe(true);
        expect(response.body.data).toBe(true);
    }, MAX_TIMEOUT);


    test("Can verify code owner", async () => {
        const electionId = await createElection(app, adminJwtToken.
            accessToken);
        const code = await createCodeForElection(app, electionId,
            userJwtToken.accessToken);
        const response = await request(app).post("/code/verify-owner")
            .send({code: code, userId: user.id, electionId: electionId})
            .set("Authorization", `Bearer ${userJwtToken.accessToken}`)
            .set("Accept", "application/json")
            .expect("Content-Type", "application/json; charset=utf-8")
            .expect(StatusCodes.OK);
```

```
45
46          expect(response.body.data).toBe(true);
47      }, MAX_TIMEOUT);
48  });
```

In Listing 16 we show an example of testing on POST requests to the /code endpoint of the API server. `jest` allows control division & execution of the tests introducing initialization and cleanup functions that will be executed before and after the execution of the tests. We can group up tests with `describe` functions, so we can better organize them among the various scenarios of the specific endpoints. `request` is a function provided by the `supertest` library that allows the creation of HTTP requests to a server. It takes as input an instance of the express application, then we can configure it specifying the type of the request and the various parameters.

We check the validity of the response by evaluating the status code and the body of the response.

7 Deployment

7.1 Prerequisites

- Unix-like operating system (either macOS or Linux);
- Docker and Docker Compose (on Mac OS make sure the file sharing implementation for the container is set to `osfx (Legacy)` (`Settings -> General`);
- Java 11 or higher;
- Node.js 18 or higher;
- npm.

7.2 Startup

To bring up the blockchain network, deploy the smart contracts on top of the peers, start the API services and the frontend web app you can use the `services.sh` script on the root of the project.

To bring up all the system's services:

```
./services up
```

Since the first time it will pull Hyperledger Fabric binaries and docker images to create blockchain artifacts and start API services, it will take a while to bring up the entire system.

While it reaches the API server creation phase, the script will execute the `npm login` command; we've preconfigured some default credentials for this purpose, these should be used when the login request is prompted:

- username: `user`
- password: `password`

Warning: During the startup phase of the API layer, the script will require the sudo privileges in order to ensure that the `verdaccio`, `cache` and `dbdata` folders have write and read permissions, which are needed on Linux environment.

The full working system consists of 33 containers (Figure 19): one for each peer and chaincode deployed on it, five containers for the API services and one for the frontend web app.

```
frontend-frontend-1                    Up 6 minutes             0.0.0.0:3000->3000/tcp
api-service-auth-server-1              Up 6 minutes             0.0.0.0:8180->8180/tcp
api-service-api-server-1               Up 6 minutes             0.0.0.0:8080->8080/tcp
api-service-redis-1                    Up 7 minutes             0.0.0.0:6379->6379/tcp
api-service-mongodb-1                  Up 7 minutes             0.0.0.0:27017->27017/tcp
api-service-verdaccio-1                Up 7 minutes             0.0.0.0:4873->4873/tcp
dev-peer1-org3-chaincode-votes_1.0-a   Up 7 minutes
dev-peer2-org3-chaincode-votes_1.0-a   Up 7 minutes
dev-peer2-org2-chaincode-votes_1.0-a   Up 7 minutes
dev-peer1-org2-chaincode-votes_1.0-a   Up 7 minutes
dev-peer2-org3-chaincode-elections_1   Up 8 minutes
dev-peer1-org3-chaincode-elections_1   Up 8 minutes
dev-peer2-org2-chaincode-elections_1   Up 8 minutes
dev-peer1-org2-chaincode-elections_1   Up 8 minutes
dev-peer1-org1-chaincode-elections_1   Up 8 minutes
peer1-org3                             Up 8 minutes (healthy)   7051/tcp, 0.0.0.0:20051->20051/tcp
peer2-org3                             Up 8 minutes (healthy)   7051/tcp, 0.0.0.0:30051->30051/tcp
peer2-org2                             Up 8 minutes (healthy)   7051/tcp, 0.0.0.0:10051->10051/tcp
peer1-org2                             Up 8 minutes (healthy)   7051/tcp, 0.0.0.0:9051->9051/tcp
peer1-org1                             Up 8 minutes (healthy)   0.0.0.0:7051->7051/tcp
rca-org0                               Up 8 minutes (healthy)   0.0.0.0:7053->7053/tcp, 7054/tcp
rca-org3                               Up 8 minutes (healthy)   7054/tcp, 0.0.0.0:7056->7056/tcp
orderer2-org0                          Up 8 minutes (healthy)   7050/tcp, 0.0.0.0:8050->8050/tcp
couchdb-peer1-org1                     Up 8 minutes             4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp
couchdb-peer1-org3                     Up 8 minutes             4369/tcp, 9100/tcp, 0.0.0.0:9984->5984/tcp
ca-tls                                 Up 8 minutes (healthy)   0.0.0.0:7052->7052/tcp, 7054/tcp
couchdb-peer2-org2                     Up 8 minutes             4369/tcp, 9100/tcp, 0.0.0.0:8984->5984/tcp
couchdb-peer2-org3                     Up 8 minutes             4369/tcp, 9100/tcp, 0.0.0.0:10984->5984/tcp
orderer1-org0                          Up 8 minutes (healthy)   0.0.0.0:7050->7050/tcp
orderer3-org0                          Up 8 minutes (healthy)   7050/tcp, 0.0.0.0:9050->9050/tcp
couchdb-peer1-org2                     Up 8 minutes             4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp
rca-org2                               Up 8 minutes (healthy)   7054/tcp, 0.0.0.0:7055->7055/tcp
rca-org1                               Up 8 minutes (healthy)   0.0.0.0:7054->7054/tcp
```

Figure 19: The set of containers that should be up and running at the end of the deployment.

To bring them down without cleaning the blockchain artifacts (it will speed up the creation of the network next times):

```
./services down
```

To clean network artifacts:

```
./services clean
```

7.3 Details

Each project's module is shipped with a script that automatizes its subpart of the system. `services.sh` script, under the hood, sequentially executes each module's deployment script in order to make the deployment of the entire system simpler.

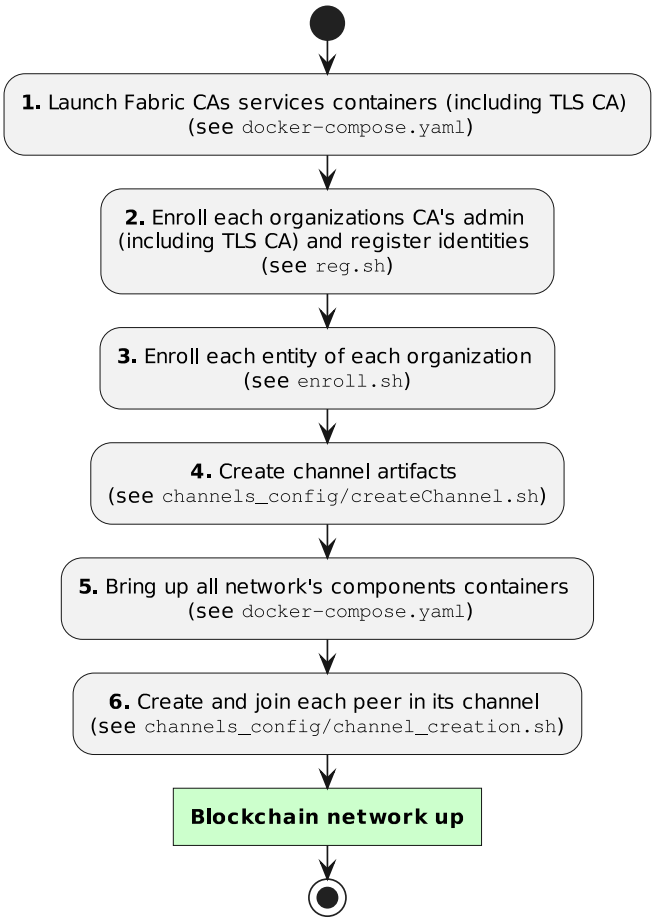


Figure 20: A sequence diagram representing the steps required to create the blockchain network.

`:blockchain` module

For this module a `network.sh` bash script automatize the deployment of the blockchain network and all its component. It has been built following the Fabric CA Operations Guide [7] [9], simulating its deployment using Docker.

The overall process is non-trivial and it is summarized in Figure 20.

After bringing up all the CA's service containers, including the TLS-CA one, (step 1) for each of them an admin is enrolled using the `fabric-ca-client` script of Hyperledger Fabric. This admin is used to register each entity into the CA database, which will be used in subsequent steps during the enrollment of these entities. These procedures are executed within the `reg.sh` script (step 2). The enrollment of each entity (both network components and users) from the Certificate Authorities (CAs) is now undertaken for each organization in `enroll.sh` (step 3). Then, the channel artifacts are generated (step 4) using the `configtxgen` script provided by Hyperledger Fabric from the configuration files

described in Section 5.1. Once the channel artifacts have been created all the network components containers are brought up, the channel is created and peers join them (steps 5 and 6).

All the network components containers are deployed through Docker Compose (using the `docker-compose.yaml` file in the `:blockchain` module folder).

`:smart-contracts` module

The `:smart-contract` module provides a set of Gradle tasks created to bring up the network, package the chaincodes and install them on the peers. These are shown below (to list them: `./gradlew tasks`):

```
Blockchain tasks
----------------
cleanAllPackages - Remove all generated packages
downNetwork - Bring down the blockchain network without cleaning
              artifacts
downNetworkAndClean - Bring down the blockchain network and clean
                      the artifacts
packageChaincodes - Build and generate chaincodes packages
upAndDeploy - Up the network and deploy chaincodes in one shot
upNetwork - Bring up the blockchain network
```

Among others, `upAndDeploy` is the Gradle task which automatizes the upping of the network and the installation of both chaincodes in one single shot.

Technically, the installation of a chaincode on peers involves, after creating the network, the following steps (Figure 21):

1. Build and generate chaincodes jars;
2. Package the jars into a `.tar.gz` files with the `peer lifecycle chaincode package` command;
3. Install the chaincodes on every peer that will endorse a transaction with the `peer lifecycle chaincode install` command;
4. After the installation, the organization's approval is needed. The set of channel members who need to approve a chaincode before it can be deployed is governed by the `Application/Channel/LifecycleEndorsement` policy described in Section 5.1;
5. After a sufficient number of organizations have approved a chaincode definition, one organization can commit the chaincode definition to the channel using the `peer lifecycle chaincode commit` command.

All of these steps have been programmed in Kotlin within Gradle tasks into the `build.gradle.kts` project's build file.

Moreover, since the startup of the network and the chaincodes installation is a very critical asset for the project a CI Pipeline has been set up to test its functioning at every pushed commit on a mirrored repository hosted on GitHub.

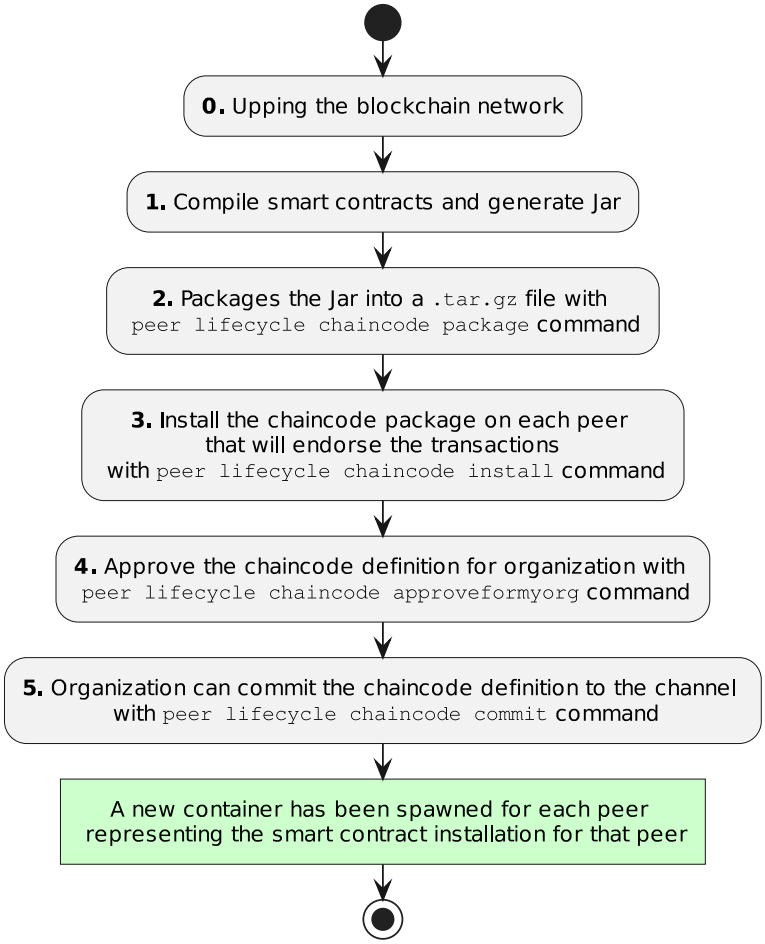


Figure 21: A sequence diagram representing the steps required to install the chaincodes on top of the peers.

`:api-service` module

We deploy all the services inside the `:api-service` module using a simple script called `startup.sh`. This contains two functions:

- `up` which will prepare the environment and start the API servers;
- `down` which will stop the API layer.

When the `up` is executed it will first copy a pair of test keys inside the `common`,`auth`,`api` folders, these are used for testing the jwt functions. After that it creates the data folders that should be used as mounting points for verdaccio and redis: `dbdata` and `cache`. After that, using the docker-compose yaml file, the `Verdaccio` server will be started and the common dependencies, inside the `:common` package will be built and published on the local registry. Once the dependencies are published the rest of the services are built and started. The `down` function will simply stop the API layer using the `docker-compose down` function.

`:frontend` module

The `:frontend` module contains just a `docker-compose.yaml` file thanks to which, simply with `docker compose up`, the frontend container can be started, listening on port `3000`.

7.4 Troubleshooting

If you are encountering issues or observing unexpected behavior during the deployment, this troubleshooting guide is designed to assist you in identifying and resolving common problems.

Linux/MacOS: `[common.tools.configtxgen] main -> Error on outputBlock: could not create bootstrapper: could not create channel group ...: open /Projects/distributed-systems/ds-project/chain-vote/blockchain/../.chainvote/blockchain/org0/orderer1/tls-msp/signcerts/cert.pem: no such file or directory`

A similar error occurs the next time a problem arises during the creation of channel artifacts because it tries to get some files inside the artifacts folder but, since the previous deployment failed, those files don't exist. Cleaning the artifacts with `./services clean` should solve the problem.

MacOS: resource management

On MacOS systems make sure to increase allocated resources in Docker Desktop, particularly for CPU and memory limit (8GBs are enough).

MacOS: `Error response from daemon: cannot stop container: <container-id>: tried to kill container, but did not receive an exit event`

Sometimes happens that, when the services are torn down (with `./services down` or `services clean`), one Hyperledger CA container remains hanging. Restarting Docker Desktop solves this issue.

MacOS: `container peer1-org2 is unhealthy`

If, during the deployment of the network, the shown message (or a similar one) appears on a MacOS host, make sure the file-sharing implementation for the containers in `Settings -> General` is set to `osxfs (Legacy)`.

MacOS/Linux: `General build failure`

While the system is starting up docker will cache lots of data that is used in the various building phases. This could lead to a general build failure in future runs. Is possible to try these solutions for solving the problems:

- `docker builder prune` to clean the docker cache;
- Delete dangling images and volumes;
- Restart docker engine;

8 Usage Examples

Once deployed the system, the web application is available on `localhost:3000` (Figure 22).

To interact with the system you have to first sign up as a new user. The only thing to pay attention to is the password, whose length must be between 8 and 16 characters and consist of at least one lowercase, one uppercase, one special character, and one number. Moreover, to allow testing the admin functionalities we give the possibility to create a new admin account − we are aware that in a real application this possibility should be avoided.

Once you are logged in as an admin you can create new elections (Figure 23)

All accounts (user and admin) are able to see the list of all elections (either open or closed) (Figure 24) and their information, including the turnout and the final results if the election is already closed (Figure 25).

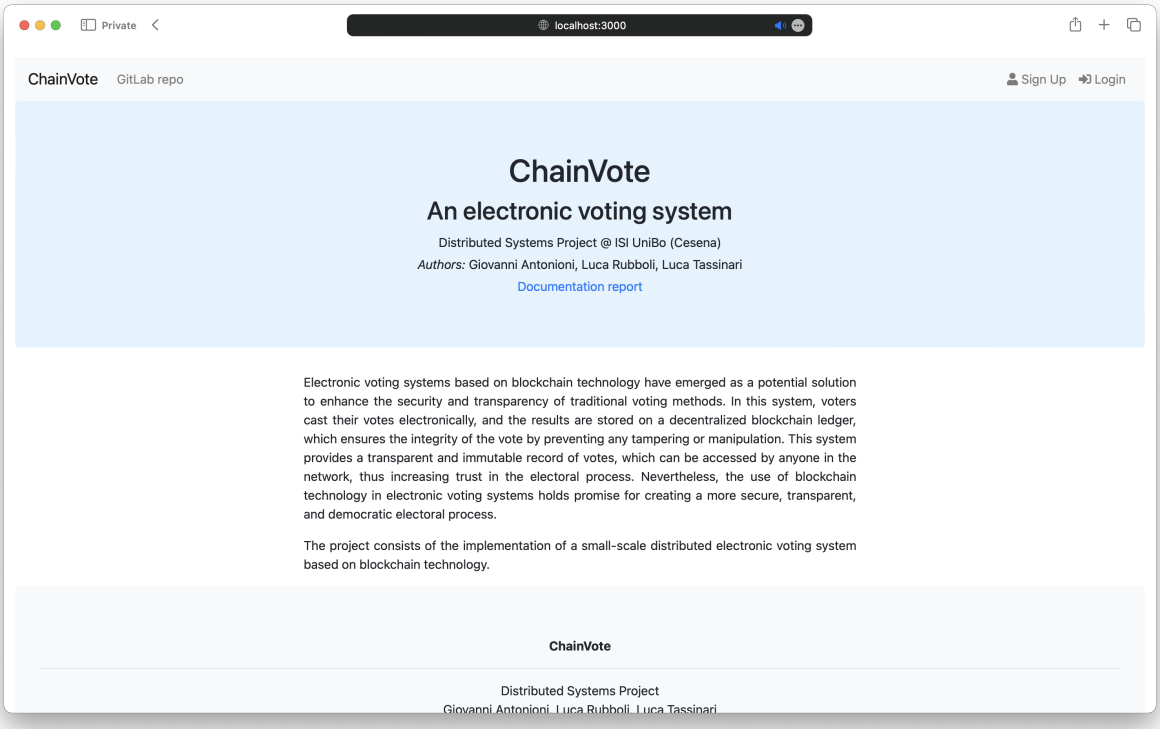


Figure 22: Homepage of the web app.

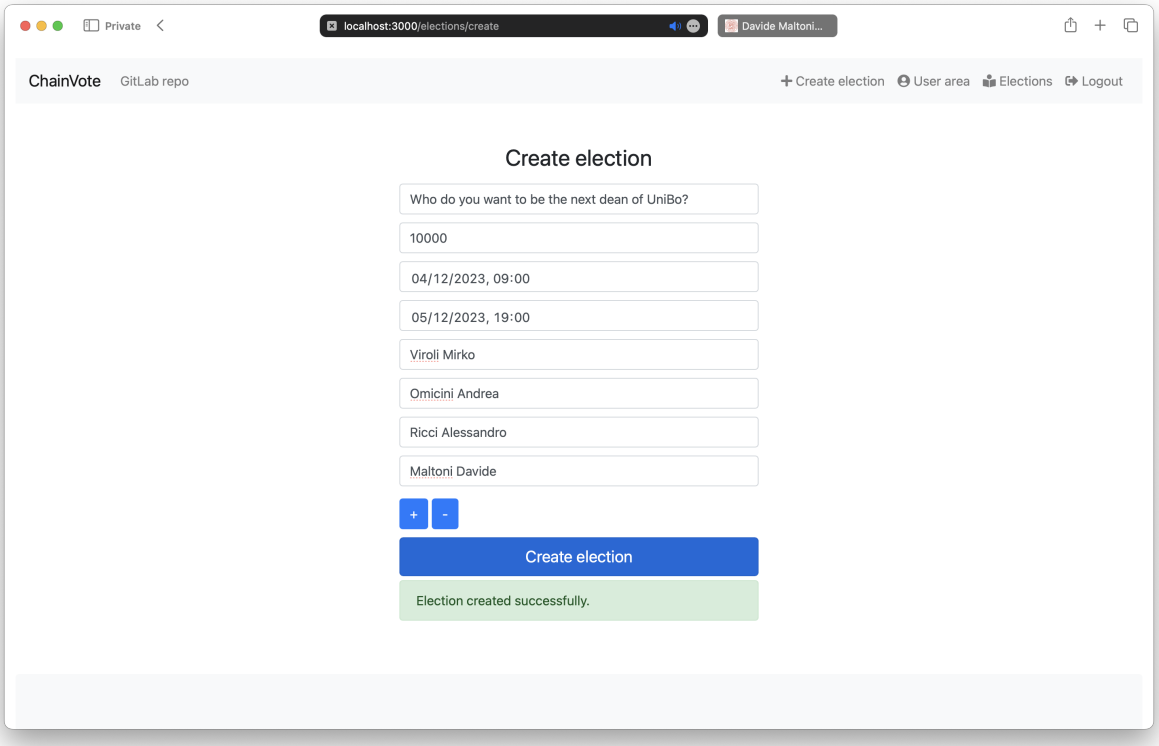


Figure 23: Page of creation of a new election.

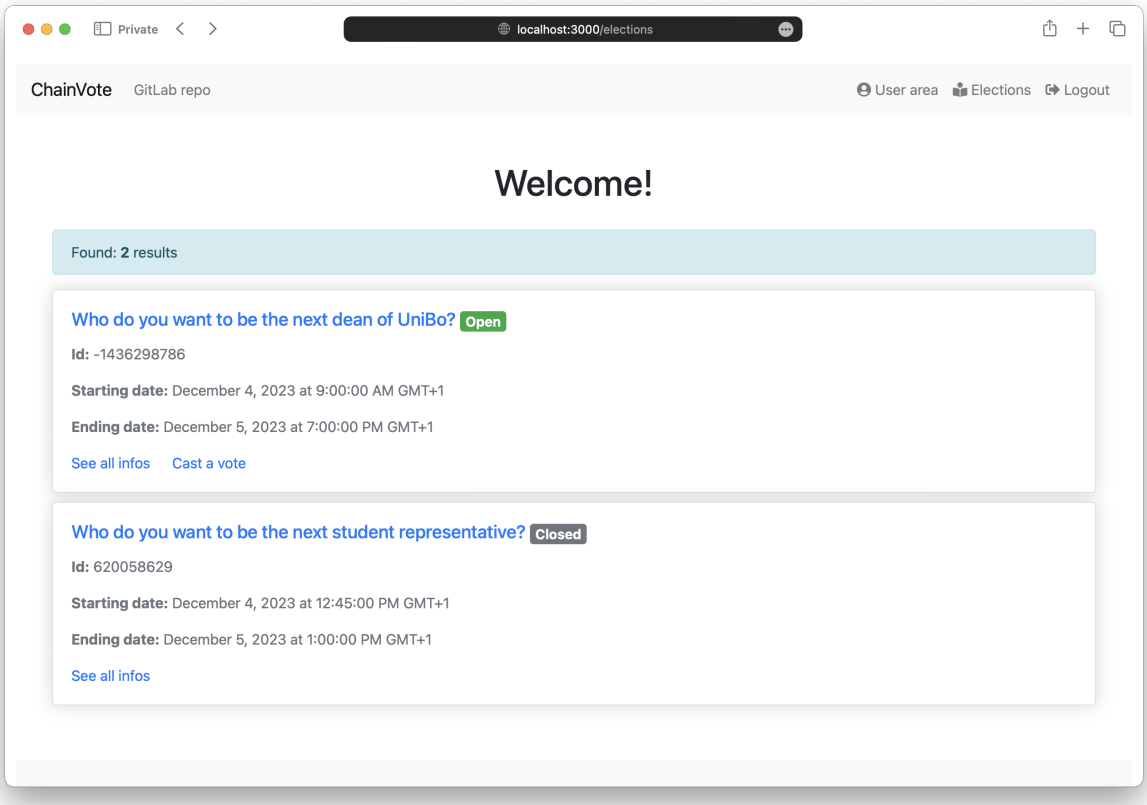


Figure 24: Dashboard with the list of all elections.

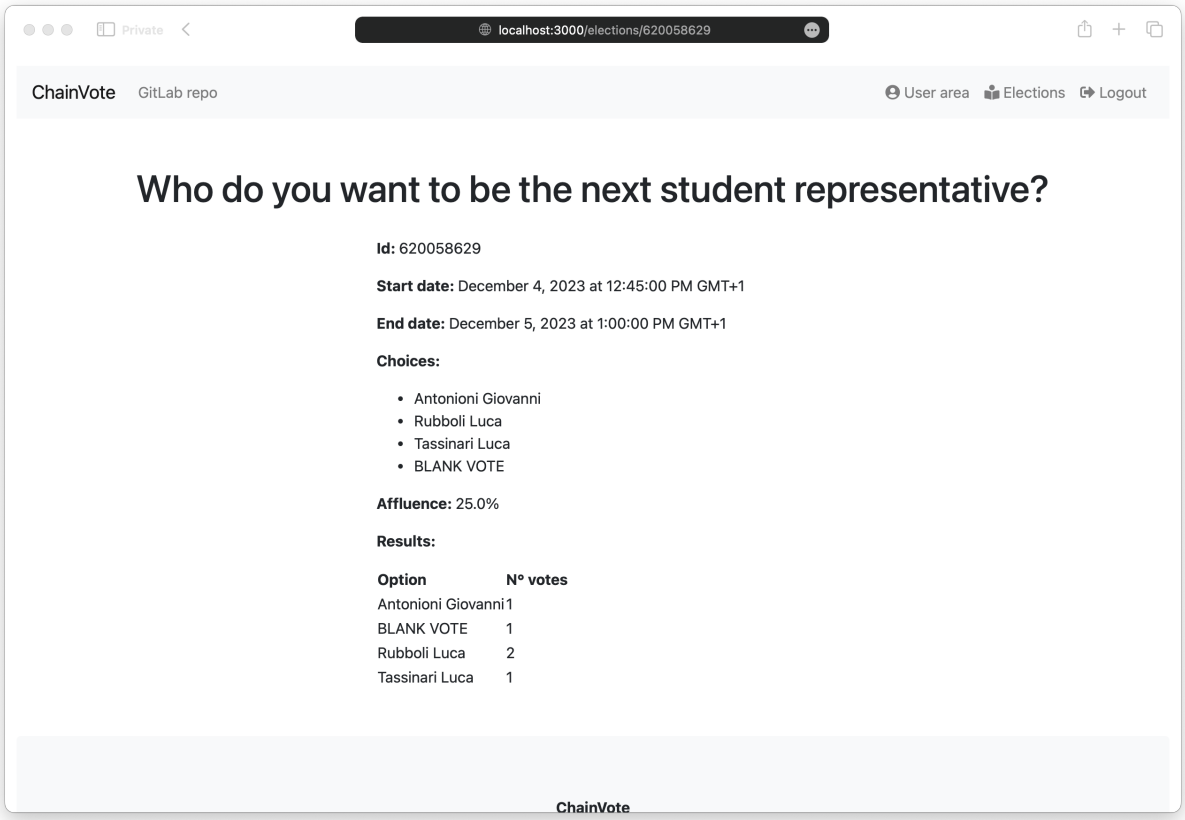


Figure 25: Page of information of a closed election, including turnout and the final results.

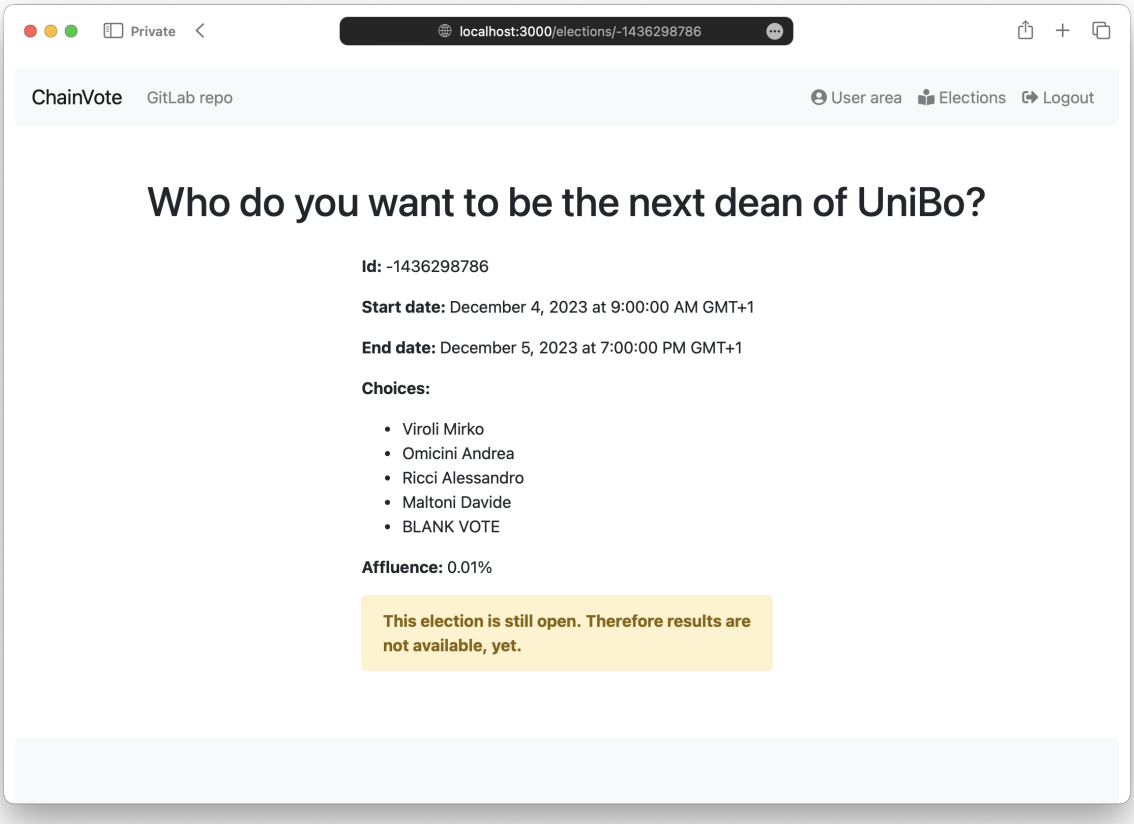


Figure 26: Page of information of an open election: only turnout is displayed.

Only logged users are able, if the election is still open, to cast a vote. At first, they have to request a new one-time code. The code, once created, is sent back to the user using the web app, including it in the appropriate input field of the page – we are aware, as already described in Section 3.1, it should be sent using a third trusted method. Then, with this code, the user will be able to cast a new vote for the election for which have requested it, indicating a preference (the user may decide to cast the vote with the requested code at a later time as well). If the user tries to request another code for the same election or submit a vote with an already consumed code an error pops up. Once the election has been correctly cast you should see the turnout has increased.

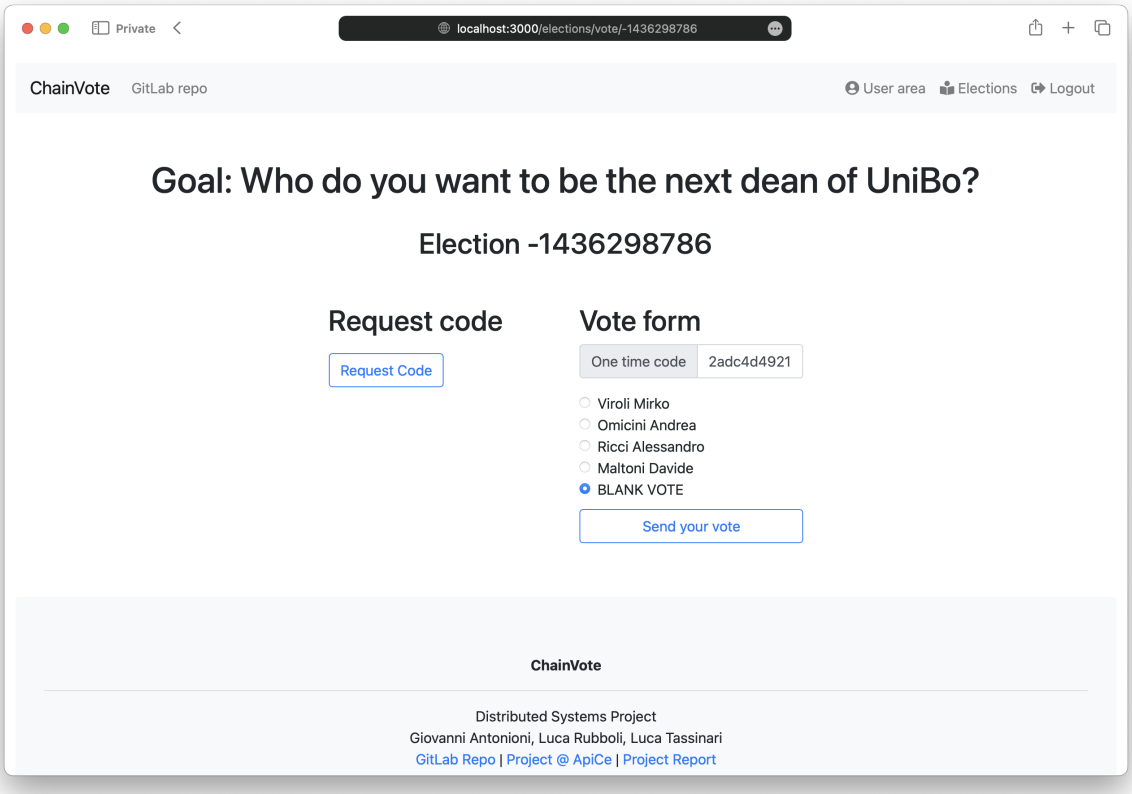


Figure 27: Page for casting a new vote.

9 Conclusions

In this project, we explored the possibility of building a reliable and secure e-voting system leveraging the advantages offered by blockchain technology.

After identifying system requirements and exploring three of the most important technologies for building permissioned blockchain, we chose Hyperledger Fabric for its mod-

ularity and high configurability.

We then proceeded with the design of the blockchain network and architecture of the project, trying to give maximum preference to its extensibility, not forgetting the three essential cornerstones for such a project: security, reliability and fault-tolerance.

Ample space was devoted to the configuration and implementation of the network and smart contracts and their deployment using Docker.

Finally, we developed a ReST API adopting best practices to provide a uniform interface, with a semantic and clear meaning, to application clients.

Despite being a challenging project we achieved the goals, fulfilling all the requirements of the system.

9.1 Future Works

To conclude, some extensions that could be implemented to make the system more comprehensive are listed below:

- As highlighted in 4.1, the blockchain network must be improved from the fault-tolerance side by building a significant amount of organization peers; this could positively affect the availability of the system too;
- As domain requirements could change, another feature could involve the ability to define customizable rules for ballots, i.e. the possibility to vote multiple choices in a single ballot, and for elections, i.e. establish criteria to invalidate results based on quorum;
- The admin creation process should be handled differently, in a real scenario an admin should be created only by means of another admin;
- Given that the process of collecting signatures for petitions to invoke a popular vote is a costly process for both organizers and participants, the system could provide a user-friendly interface to handle proposals and automatically generate a referendum when a quorum is reached;
- Tests could be enriched by simulating a wider variety of stress situations;
- From a security point of view, the mechanism used to provide the code to users should rely on a secure fashion, i.e. a combination of out-of-band channels.

References

[1] Apache couchdb. `http://couchdb.apache.org`.

[2] Google leveldb. `https://github.com/google/leveldb`.

[3] Hyperledger fabric. `https://www.hyperledger.org/projects/fabric`.

[4] Auth0. Json web tokens specification. `https://datatracker.ietf.org/doc/html/rfc7519`, 2023.

[5] Hyperledger Fabric Documentation. Hyperledger fabric documentation, 2023.

[6] Nitin Gaur, Anthony O'Dowd, Petr Novotny, Luc Desrosiers, Venkatraman Ramakrishna, and Salman A. Baset. *Blockchain with Hyperledger Fabric: Build decentralized applications using Hyperledger Fabric 2, 2nd Edition.* Packt Publishing, 2nd edition, 2022.

[7] Hyperledger Fabric CA Documentation. Hyperledger Fabric CA Operations Guide. `https://hyperledger-fabric-ca.readthedocs.io/en/latest/operations_guide.html`, 2023.

[8] JUnit Team. JUnit. `https://junit.org/`, 2023.

[9] KC The Servant. Rework: A Companion Guide to Fabric CA Operation Guides for Fabric v2.2. `https://kctheservant.medium.com/rework-a-companion-guide-to-fabric-ca-operation-guides-for-fabric-v2-2-7886e8037427`, 2020.

[10] Mockito. Mockito: Mockito framework. `https://github.com/mockito/mockito`, 2023.

[11] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[12] R3. Corda: An open-source blockchain platform for business. `https://www.corda.net/`, 2023.

[13] Tendermint Team. Tendermint: Byzantine Fault Tolerant Middleware. `https://tendermint.com/`, 2023.