Security and Privacy Modeling, Implementing with XACML/ALFA and Fabric

Submitted By

Venkata Satya Siddhartha Illa

Venkata Naga Bhaavagni Maddi

Farhana Begum Shaik

Sripadh Vallabh Kaparthi

Ganesh Nyaupane

Under the Instructions of

Dr. Kewei Sha

Under the Guidance of

Dr. Kwok-Bun Yue

Dr. Wei Wei

Joses Selvan

Kayaanoosh Collector

DEPARTMENT OF SCIENCE & ENGINEERING

UNIVERSITY OF HOUSTON – CLEAR LAKE

2700 Bay Area Blvd

Houston, Texas-77058

Abstract

The primary goal of this project is to implement more sophisticated and flexible access control model using the ABAC methodology suitable for the requirements of the project developed in the Hyperledger Fabric framework. ABAC is a logical access control methodology where authorization to perform a set of transactions is determined by evaluating attributes associated with the subject, object, action and in some cases, environment conditions against policy, rules, or relationships that describe the allowable transactions for a given set of attributes. In this project, we identified few ABAC attributes from the BPMN work-flow diagram, and these attributes are assigned to a specific identity. Our goal is to add these attributes for specific users to provide the access control. According to our project requirement there are three different kinds of MBSE Assets defined, they are Change Request, BCDocument and DocumentPackage and we use these asset data for validation. All the security and privacy policies are defined in ALFA, which is a programming language used for writing the access control policies. Here, all the security policies are implemented in the form of Smart Contracts, and we accessed them to authorize a particular transaction. When a policy is successfully executed, it means the user is having the right privilege and he can submit the transaction successfully. When a request is made for an asset, it goes through the approval process and based on the level of access, the appropriate security policy is implemented thus the user will be able to do operations making changes to the world state and hence providing the necessary security check. In this way, we are achieving Access control based on attributes and thus have granular level of control on the transactions. For implementing the security policy, we are following the static struct which is having a null value problem. In the future, this static struct can be replaced with a dynamic struct for a better enhancement.

Table of Contents

1. Introduction

Attribute-based access control, also known as policy-based access control is a paradigm where access rights are granted to users through the use of policies which combine attributes together. As the access is established based on the attributes, there will be higher levels of access security, beyond providing the access solely based on the roles. An Identity will be generated in the MSP Folder and stored in the in-memory wallet upon enrollment by the admin user with some required ABAC attributes. The CA generates a certificate which have all the attributes that are enrolled and obtains a digital certificate from a given CA and registration is done by a registrar in which a CA will issue the digital certificate. In this project, we developed an accurate and complete set of security policies written in ALFA using the required ABAC attributes for the Gateway MBSE project that is capturing a majority of the MBSE project requirements making use of blockchain assets like Change Request, BCAssetMetaData and DocumentPackage. All the ABAC policies are developed and implemented in Smart Contracts using GOLANG.

1.1 Scope of the project

Consortium blockchain applications usually have stringent and nuanced security requirements. Attribute-Based Access Control (ABAC) is the most flexible security model suitable for the requirements. Once the security attributes and policies are specified, we implement the policies using the Smart Contracts. We are configuring organizations and users, and their attributes in Fabric's MSP, and setting up the configuration corresponding JSON files accordingly. Initially, an Admin is registered, then a user gets registered and enrolled with the required attributes. All the ABAC attributes are analyzed and retrieved from the BPMN diagram. In addition to that, the security policies are written in ALFA, which then are implemented using the Smart Contracts.

1.2 Blockchain Network

There are four types of blockchain structures:

1.2.1 Public blockchain: In this type of blockchain network, anyone with network connectivity may join on a blockchain platform and become an authorized node, making public blockchain non-restrictive and permissionless. The user has access to current and historical data, as well as the ability mining operations, which are complicated calculations required to validate transactions and add them to the ledger. On the network, no legal record or transaction can be modified, and because the source code is typically open source, anybody can check the transactions, identify errors.

1.2.2 Private blockchain: They are controlled by a single organization and central authority determines who can be a node. While it performs similarly to a public blockchain network in terms of peer-to-peer connectivity and decentralization, this blockchain is substantially smaller. For example, in this blockchain write permissions are kept to NASA, and read permissions may be public or restricted to certain participants like UHCL and Tietronix.

1.2.3 Consortium blockchain: A Consortium blockchain is one in which the consensus process is controlled by a pre-selected group of nodes. In this project, the team used a Consortium blockchain network that has nuanced security requirements and ABAC is the most flexible security model suitable for the requirements. The ability to read the blockchain may be open to the public or limited to participants. It reduces transaction costs and data redundancies, simplifying document handling and getting rid of semi-manual compliance mechanisms.

1.2.4 Hybrid blockchain: This type of blockchain is used by an organization that needs both elements of a private and public blockchain. It allows an organization to create a private blockchain, allowing them to manage who has access to certain data. Transactions and records are typically not made public, but they can be validated, when necessary, for as by granting access through a smart contract.

1.3 Hyperledger Fabric

1.3.1 Prerequisites to install Hyperledger Fabric

To install Hyperledger Fabric v2.3 on the Linux machine, first, we need to install Git and CURL after we need to install Docker and Docker Compose, which is a tool for defining and running multi-container Docker applications. With Docker Compose we can configure the YAML file to configure the application's service.

To make sure that the Docker is running or not, we use the command *sudo systemctl start docker*. After setting up all the installations, there is a need for samples and binaries installation. Then, determine a location on the machine where to place the fabric-sample repository and enter the directory in a terminal window. The command that follows will perform the following steps:

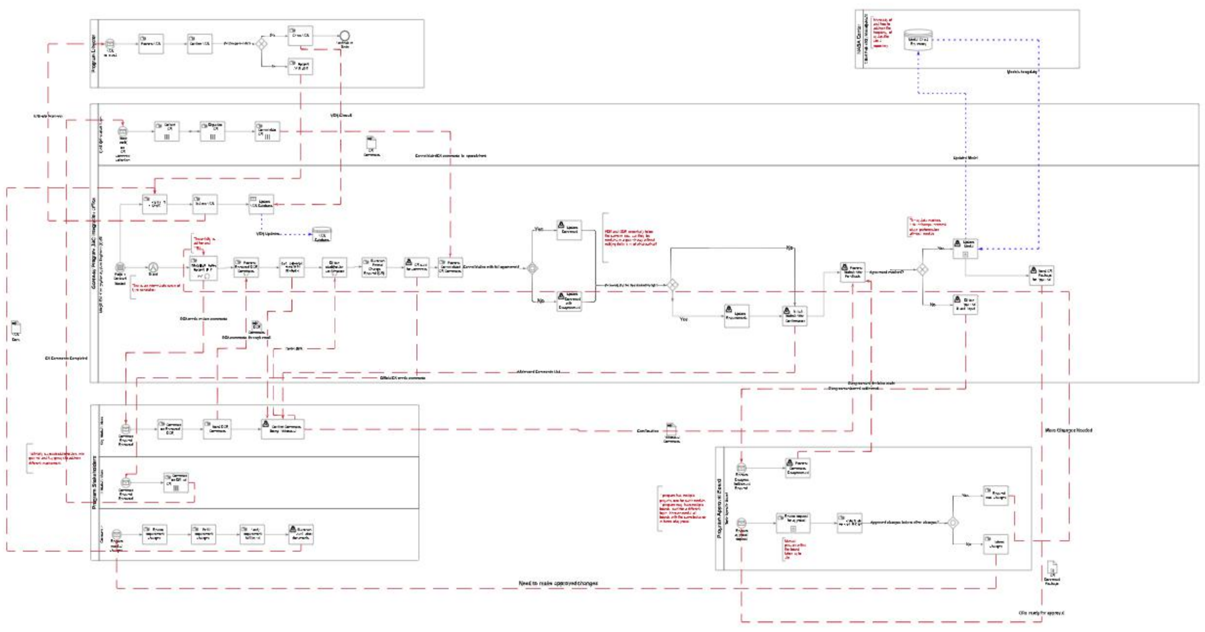
a. If needed, clone the Hyperledger/fabric-samples repository
b. Check out the appropriate version tag

c. Install the Hyperledger Fabric platform-specific binaries and config files for the version specified into the /bin and /config directories of fabric-samples

d. Download the Hyperledger Fabric docker images for the version specified

Once everything will ready, and in the directory into which we need to install the Fabric Sample and binaries.

1.4 Approval process



Explanation:

Participants:

- Gateway Program Director
- Gateway Program JSC Integration Office
- Gateway Program Stakeholders
- Gateway Program Approval Board
- NASA Cloud

Initially MBSE Chief Integration System Engineer (CISE) initiates a Draft copy of the change request and sends an email to Key Stakeholders requesting to verify and write comments on it. After receiving the comments through the email, change request process the received DCR comments and send to obtain stakeholder confirmation. Obtaining stakeholder confirmation is an iterative process until the process completes successfully. Next it is sent to create a Formal Change Request which is stored in Blockchain. Then Change Request sent for comments for All Stakeholders (Program Stakeholder). It receives the comments and make it official and sent to CISE office staff team where they collect, organize, and consolidate the Change Request and send to process consolidated CR comments.

If every stakeholder and CISE staff team fully agree with the CR, then it will update the document to prepare for submission of approval, else it updates the document with disagreement notice. Next if there is no further update, then it is sent to solicit stakeholder confirmation, else it updates requirements and sends. Then the change request is sent to the Process Stakeholder. If agreement is yes here, then it needs to update the model and send a package for approval. Here the Change Request is ready for approval and sent to the Program Approval Board.

Now in the Program approval board, once it receives the requests, it reviews and communicates approved changes. If it needs more changes the process is repeated for new change requests otherwise it informs changes and sent to contractors to review, fulfill, verify and generate verification documents and send them to VCN (Verification Confirmation Notes). Here for every requirement program which relies on the contractor, they must come back and show proof that they met the requirement only then VCN is approved.

Next, Submitted VCN is sent to the Program Director. Here it processes and confirms VCN. If it is successful, then it performs the VCN close, and the process ends. If it is unsuccessful, it requests for modification and repeats the steps again till the VCN is closed. After submitting the VCN, it updates in the VCN, it updates in the VCN Local Database.

1.5 Wallet

A wallet contains a set of user identities. An application run by a user selects one of these identities when it connects to a channel. Access rights to channel resources, such as the ledger, are determined using this identity in combination with an MSP.

The three different types of wallet storage: File system, In-memory and CouchDB.

File System: This is the most common place to store wallets; file systems are pervasive, easy to understand, and can be network mounted. They are a good default choice for wallets.

In-memory: A wallet in application storage. Use this type of wallet when your application is running in a constrained environment without access to a file system, typically a web browser. It's worth remembering that this type of wallet is volatile; identities will be lost after the application ends normally or crashes.

CouchDB: A wallet stored in CouchDB. This is the rarest form of wallet storage, but for those users who want to use the database back-up and restore mechanisms, CouchDB wallets can provide a useful option to simplify disaster recovery.

1.6 Blockchain Assets:

There are few block chain assets which are implemented in the project. Some of the attributes are as follows, BCAssetId used as an identifier, BCAssetType can be Change Request, Document and Document Package. In the same way, there is a project attribute that internally has orgRoles. CRSubmissionTime, IsWithdrawn, WithdrawnTime, CRDecisions and CRComments are few attributes to update when a policy is implemented.

1.7 Abbreviations

ABAC	–	Attribute Based Access Control
ALFA	–	Abbreviated Language for Authorization
CA	–	Certificate Authority
CC	–	Chain Code
CR	–	Change Request
DCR	–	Draft Change Request
CISE	–	Chief Integration System Engineer
MBSE	–	Model Based System Engineer
SC	–	Smart Contract
MSP	–	Member Service Provider
VCN	–	Verification Closure Notice
XACML	–	eXtensible Access Control Markup Language

2. Requirements

2.1 Functional Requirements

- Understanding the BPMN Workflow to identify the potential ABAC attributes
- Implementing the ALFA policies based on the project requirements for approval and collaboration process
- Registering the user and creating the identities
- Adding/Updating ABAC attributes for a particular user
- Storing the Certificates that are in the MSP to the Postgres database and in-memory wallet
- Storing and accessing the credentials and attributes to the external database (Postgres) and in-memory wallet
- Invoking a Smart Contract based on the user's transaction
- Implementing Security policies using Smart Contracts in GOLANG
- Permitting user access to the Application depending upon the role to the world state using appropriate smart contracts

Implement Certificate Authority (CA)

- Initialize Fabric-CA Server
- Generates Certificates for the members, peers to the network using respective organization's CA's
- Attributes are stored and retrieved

Implement Membership Service Provider (MSP)

- Defining organizations that are trusted by Fabric network
- Creating Local MSPs for peers of the organizations
- Creating Global MSPs for the network and organizations
- Storing attributes of the peers, channels, network using the database and defining the policies for the membership to the network
- Enabling entities to access permissioned Blockchain

2.2 Software Requirements

ALFA: Programming Language used to write access control policies.

Angular 14: Angular is an application design framework and development platform for creating single-page apps. AngularJS extends HTML attributes with Directives and binds data to HTML with Expressions. It changes the static HTML to dynamic HTML.

Curl: Open-source software used for transferring data in a CLI.

Docker: Open platform for developing and running applications. Used for separating applications from the infrastructure.

Docker-compose: Compose is a tool for defining and running multi-container Docker applications. With compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

PostgresSQL 14: Postgres is a free and open-source relational database management system emphasizing extensibility and SQL compliance Postgres is used as the primary data store or data warehouse for many webs, mobile, geospatial, and analytics applications.

Visual Studio Code: Visual Studio Code is a source code editor that can be used with a variety of programming languages. Visual Studio Code combines the simplicity of a source code editor with powerful developer tooling, like IntelliSense code completion and debugging.

Git: Git is a free and open-source version control system used to handle small to very large projects efficiently, Git is a software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development

GitHub: GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

Node 16: Node.js is an open-source, cross-platform, back-end JavaScript runtime environment and executes JavaScript code outside a web browser. It is used for server-side programming, and primarily deployed for non-blocking, event-driven servers. It's used for websites and back-end
API services.

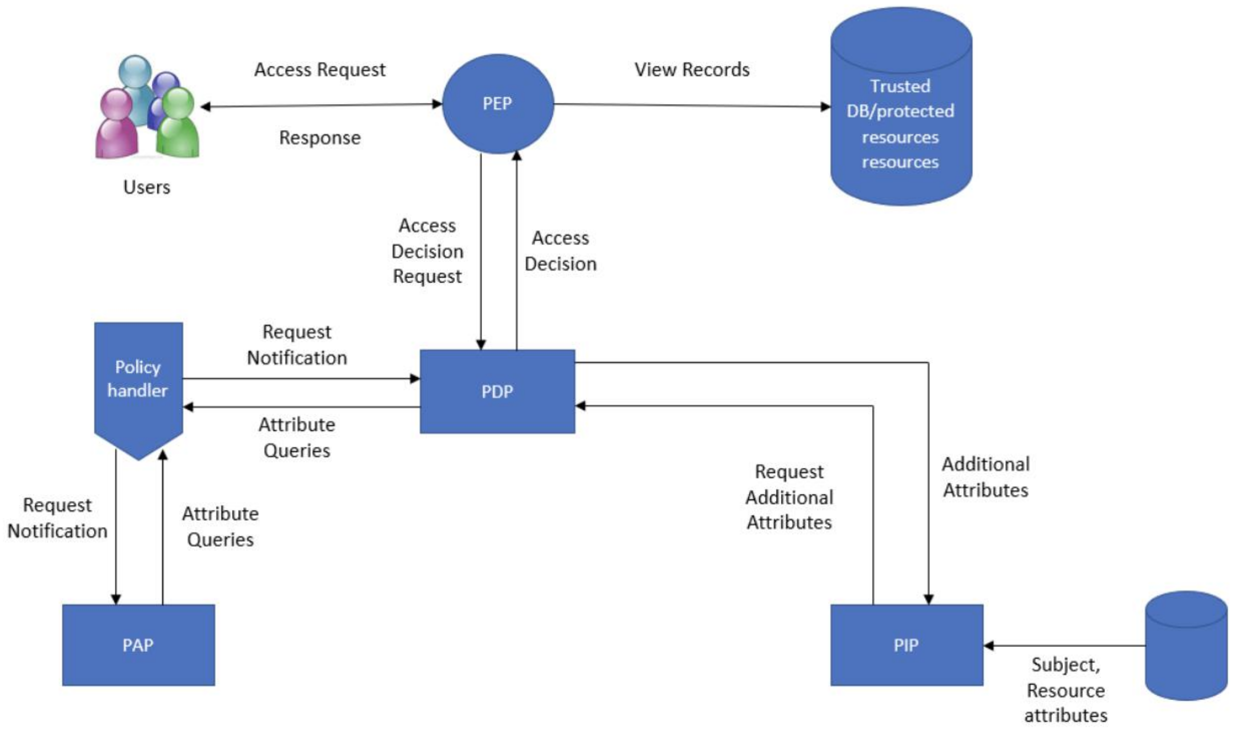
Go: Go is a statically typed, Scalable, faster execution and a compiled programming language.

2.3 Hardware Requirements

Operating System	-	Ubuntu 20.04.1 and Windows 64.bit
Processor	-	2GHz dual core
System Memory	-	4 GB RAM
Hard Drive Space	-	50 GB
Graphics	-	VGA capable of 1024x728 resolution
USB or CD/ DVD Drive	-	At Least one available for installer media
Internet Adapter	-	Wired or Wireless Network

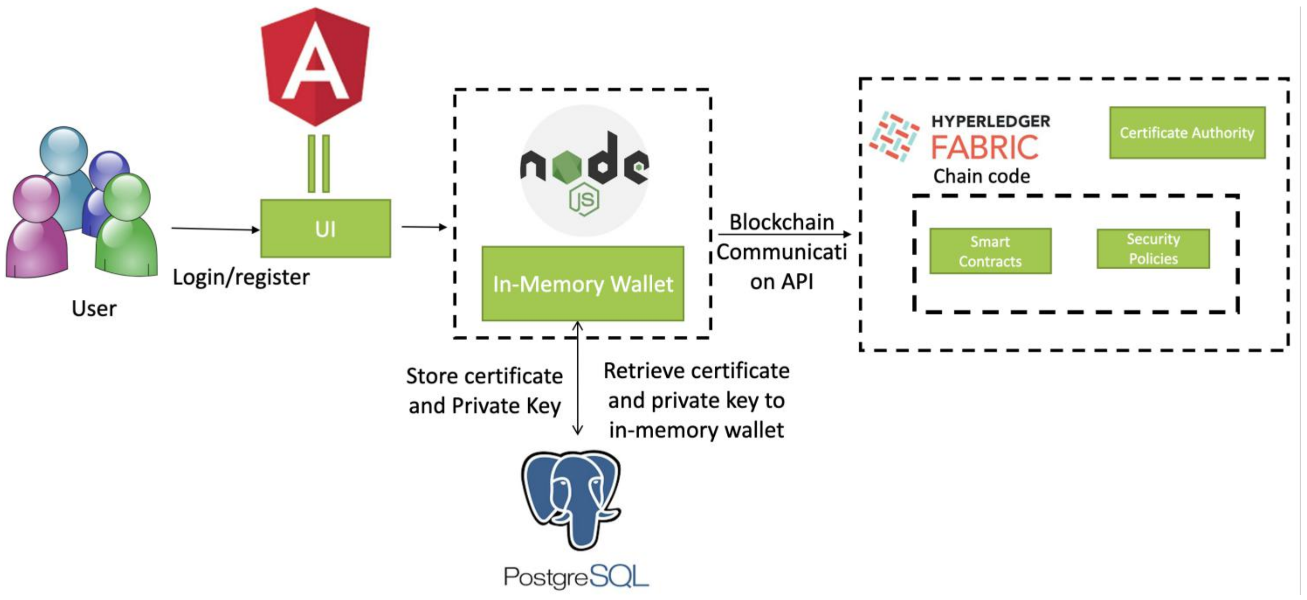
3. Design

3.1 ABAC Architecture



When a user sends an access request to the authorization system, the request is handled by the Policy Enforcement Point (PEP). It will then convert those requests into XACML authorization for making a decision. So, this converted request is transferred to the Policy Decision Point (PDP) where it evaluates the authorization request by checking with predefined policies in the policy handler as shown in the above figure. All the policies are managed by the Policy Administrator Point (PAP), which is responsible for addressing attributes queries and requests. If needed PDP also retrieved attributes values from Policy Information Point (PIP), it has all subject, resource, environment, and action attributes. After, PDP decides the form of Permit/ Deny/ NotApplicable/ Indeterminate and returns to the user via PEP. Therefore, authorization flow has mainly 4 components.

3.2 Architecture Flow

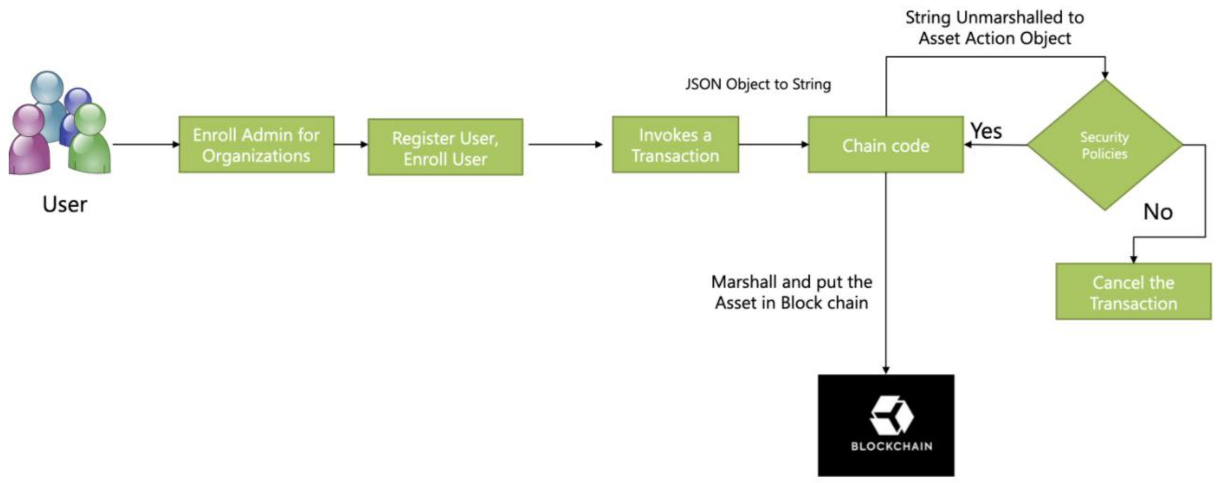


The Architecture flow of this project consists of:

- Front End / UI: Angular
- Server: Node Js
- Database: PostgreSQL
- Blockchain: Hyperledger Fabric

A user initially can register/login in the Front End/UI application. Later few API's will be called from Node Js server at the time of user registration to interact with the MBSE Application. User's certificate and private key are generated from the Certificate Authority component and are stored in the postgresSQL Database and are retrieved to the in-memory wallet. The retrieved data is checked for the authentication with MSP. We make use of Smart Contracts to implement the Security Policies. The Security Policy determines whether the transaction is authorized or not by verifying the user's identity attributes along with the AssetAction attributes. If both the client identity and the asset action attributes meet the security policy conditions, then a particular transaction is authorized. A Security policy is returning a Boolean value as a token of security check. Based on the Boolean value, the smart contract accepts or rejects a transaction. If the smart contract accepts the transaction, then the AssetAction object is marshalled into array of bytes, which is then written to the world state.

3.3 Data flow


String Unmarshalled to Asset Action Object
JSON Object to String
User
Enroll Admin for Organizations
Register User, Enroll User
Invokes a Transaction
Chain code
Yes
Security Policies
No
Cancel the Transaction
Marshall and put the Asset in Block chain
BLOCKCHAIN

The Data flow of the project is as follows. Initially, an Admin is registered for an organization. Then admin registers the users and gets the client secret after successful registration. Then the user enrolls himself with the client secret. After the successful registration and the enrollment of the user. The user can invoke the transaction based on his privileges. The user invokes a transaction with an AssetAction JSON Object. We convert the JSON object to a string using json.stringify(). Internally when a chain code is called, a JSON Object needs to be converted to string before sending over the network. Inside the SC/CC the string is unmarshalled to AssetAction Object. Based on the AssetAction Object attributes like ActionId and AssetType, we determine the corresponding smart contract that has to be invoked. When the appropriate smart contract is invoked, we verify the client users authorization with the help of a Security Policy. The Security Policy determines whether the transaction is authorized or not by verifying the client identity attributes along with the AssetAction attributes. If both the client identity and the asset action attributes meet the security policy conditions, then we authorize that particular transaction. If the transaction is authorized, we are returning a Boolean value True else if the transaction is not authorized, we return a Boolean value False. Based on the Boolean value, the smart contract accepts or rejects a transaction. If the smart contract accepts the transaction then the AssetAction object is marshalled into array of bytes, which is then written to the world state.

4. UML Diagrams

4.1 Model UML Diagrams

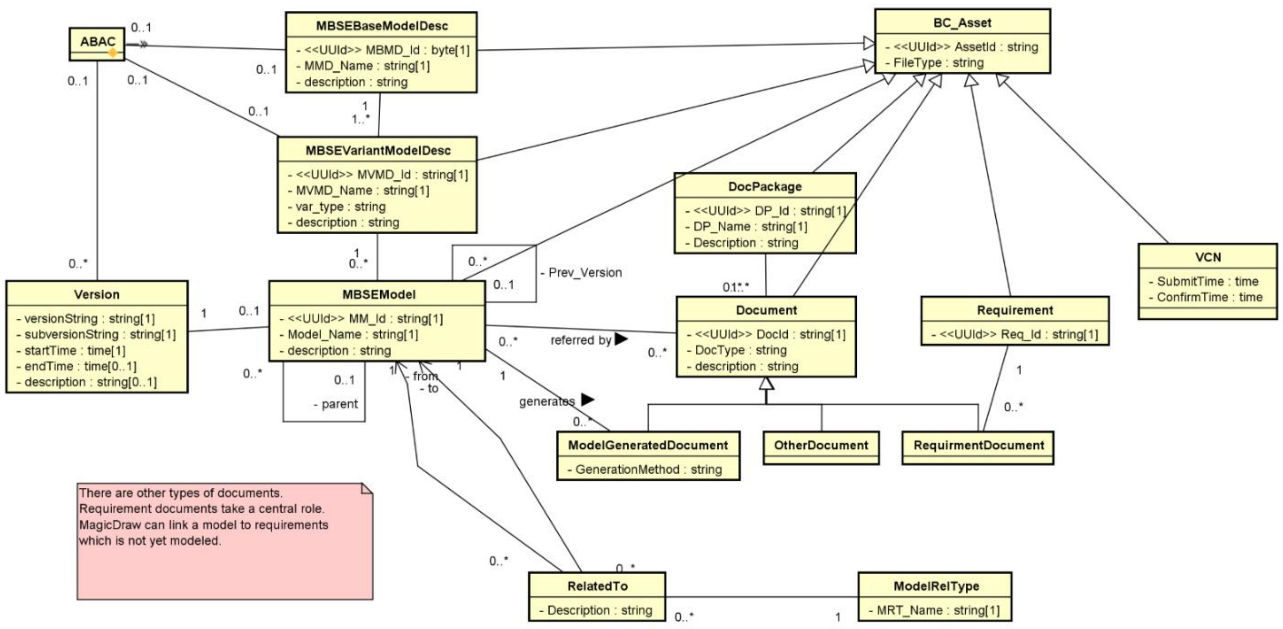
A Project high-level class diagram showing major classes and their associations. Organization, projects, roles, and employees in this class diagram are business entities satisfying business requirements. A Model high-level class diagram showing major classes and their associations.

Three major Models are:

a. Class MBSEBaseModelDesc

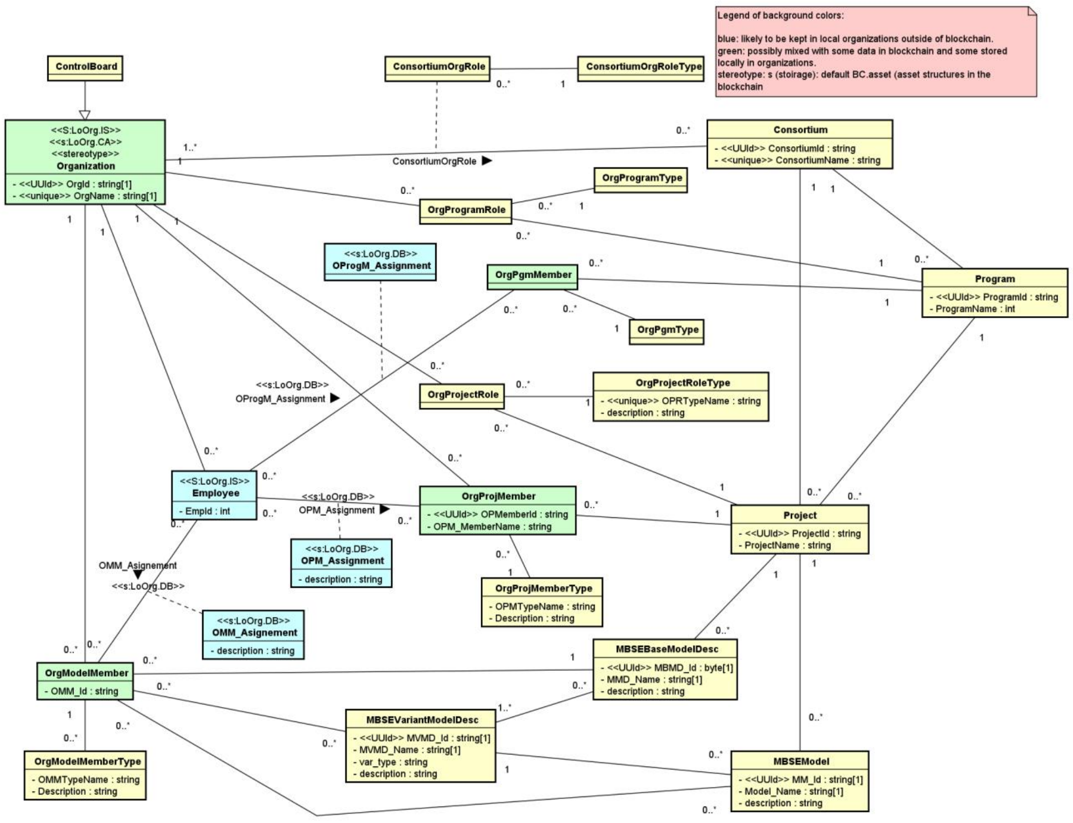
b. Class MBSEVariantModelDesc

c. Class MBSEModelDesc



4.2. Relational Schema Diagram and Blockchain Asset JSON Schema

This relational schema provides the information about relation between Consortium to different organizations and relation to the Project.



5. Components

5.1 Identities

The different actors in a blockchain network include peers, orderers, client applications, administrators and more. Each of these actors are active elements inside or outside a network able to consume services has a digital identity encapsulated in an X.509 digital certificate. These identities really matter because they determine the exact permissions over resources and access to information that actors have in a blockchain network.

A digital identity furthermore has some additional attributes that Fabric uses to determine permissions, and it gives the union of an identity and the associated attributes a special name principal. Principals are just like userIDs or groupIDs, but a little more flexible because they can include a wide range of properties of an actor's identity, such as the actor's organization, organizational unit, role or even the actor's specific identity.

For an identity to be verifiable, it must come from a trusted authority. A membership service provider (MSP) is that trusted authority in Fabric. More specifically, an MSP is a component that defines the rules that govern the valid identities for this organization. The default MSP implementation in Fabric uses X.509 certificates as identities, adopting a traditional Public Key Infrastructure (PKI) hierarchical model (more on PKI later).

5.2 Attributes

In an attribute-based access control system, attributes are used to determine access through any type of attributes such as subject attributes, resource, action, and environmental attributes. In Gateway Program Model Management and Approval Process Collaboration Diagram, the team considers several domains, with each domain including several entities (e.g., Gateway Program JSC Integration Office as Department, NASA, and UHCL as Organization).

In the below table, **Subject attributes** for the simplified version of BPMN design are Organization, Organization Unit as they are initiated and process the change request. Change request and its employee ID's role should be involved. Project Name and program is also considered as subject attribute.

Resource attributes could be Owner of the project, Status of the project, Asset type as an organization have many assets, project name, and the model such as here CR Approval is the project which is considered under Gateway model.

When requesting any CR document, the request will be checked as to what date/time it arrived, which location and IP address, which type of communication channel is used, and many other entities. When CISE starts the project contract, that process should store some information regarding the date, time, device IP/MAC address through which it accessed, location.

Lastly, the **action attribute** defines what the user wants to perform. In simplified BPMN design, suppose all stakeholders want to access the Change Request document to comment, the action attribute may be described as read and update. CISE has all rights on CR documents, Modules, and requirements, so CISE "action type = read, update or delete".

Subject Attribute	**Example**	**Notes**
Organization	UHCL; NASA	Main unit who is involved in the process of change request
Organization Unit	JSC Integration Office	Under this unit all the requirements, CRs, and modules initiate and process
Role	CISE, Staff Team, Program Director	Who are involved in the process of the CR documents
Security Clearance	Top Secret; Secret; Classified	Security information
Employee ID		
Project Name	CR Approval	Project name for the approval
Program	Gateway approval	Name of the program
Resource Attribute		
Owner	NASA; JSC	Have Change

		Requests/Comments
Status	Official; Updated (and update with agreement/disagreement)	CR may be updates or official
Asset Type	CR Module, Comments, Requirements	Organizations have many assets
Model		Who can access on which model
Project Name	CR Approval	Name of the project in the network
Action Attribute		
Read, Update, delete	CISE	Initiate, generate and process and make comments
Read	Staff Team	Collect, organize, and consolidate
Read, Update	Stakeholders	Comments on receiving CR and confirm

Attributes for the approval process

5.3 Policies

5.3.1 Specification of Security Rules, Policy, and Policy Sets

Allowable activity inside an organization is defined by Policy, Rules, and relationships, which are based on subject privileges and how resources are to be protected under environmental attributes. ABAC security rules are constructed with four categories subjects, resources, environment, and action.

By providing characteristics and literals to compare with, the target describes the component's subject(who), action(what), and resource(which). Because policies inside a policy set and rules within a policy might return various results, the result is derived by a combining algorithm to combine the separate decisions.

Related to simplified BPMN design, suppose a policy state that “all employee belonging to the CR approval project should have access to requirements sent to all stakeholders within the program stakeholder to which they belong”. A policy can be a basic authorization rule (permit | deny target: access document) and a policy set can be (permit target: equal (action/id, read) and equal (subject/role, employee)).

If a policy set contains multiple policies (and policy sets) and those policies return different decisions, then a combining algorithm will be applied. They are used to decide by different children of parent policy into a single decision that the given policy will return to its parent. There are two types of combining algorithms: policy combining algorithm and rule combining algorithm. There are also many algorithms and some of them are used in this project such as Deny overrides, Permit overrides, First applicable, Only one applicable, ordered-deny-overrides, and ordered-permit-overrides and Deny unless permit and Permit unless deny combining algorithms.

5.4 Certificate Authority

A **Certificate Authority (CA)** is utilized to validate the computerized characters of the users, which can go from people to PC frameworks to servers. Certificate Authorities protect the system against fake entities and handle the life cycle of any number of digital certificates.

5.4.1 Concept of Public key and Private key

A **public-key** certificate is a digitally signed document that confirms the sender’s identity and authorization. It employs a cryptographic framework that associates a public key with a specific entity, such as a user or company. A certification authority, a trustworthy third party, creates and issues the digital document. **Public key** certificates include a public key, identity information about the owner, and the name of the issuing certificate authority (CA). A message encrypted with the public key can only be decrypted with the corresponding private key. **The private key** is used by the recipient to decrypt a message that is encrypted using a public key. Since the message is encrypted with a specific public key. It can only be decrypted with the corresponding private key.

5.4.2 Root CA and Intermediate CA

A root certificate authority (CA) is a trusted CA that has the authority to validate an individual's identity and sign the root certificate that is delivered to the user. Because it has been validated and signed by a trustworthy root CA, the certificate is regarded as acceptable. Between the root CA and client certificate that the user enrolls for, an intermediate CA acts as a link. Certificates that are generated from the intermediate CA are trusted if they were signed by the root CA.

5.4.3 Certificate Chain

A Certificate Chain is a multi-leveled hierarchy of trust and is a collection of certificates that starts with a server certificate and ends with the root certificate. We can follow the chain from the client's certificate to a single root CA, and each link leads to a person (or organization) who is ultimately responsible for all the trust.

The signature of the server's certificate must be traced back to its root CA if it is to be trusted. Every certificate in the chain is signed by the entity whose identity is determined by the next certificate in the chain. Trusted root CAs are a group of CAs that are automatically recognized by clients. Meanwhile, server and intermediate certificates might be signed by a CA that the client does not recognize. The root CA may then sign the intermediate CA, which could then issue the server certificate. When a client connects to a server with a certificate signed by intermediate CA, the server's certificates may now be traced back to the root certificate via an intermediate certificate, and the client can trust the server.

This whole specifies the key management and certificate monitoring by combining multiple CAs into a hierarchy structure, where the root CA automatically verifies the whole chain.

5.4.4 Server and Client CA

CA Servers can process user certificate enrollment requests, issue, and revoke digital certificates. All CA servers are designed to meet the needs of identity management. Organizations may effectively ensure their user's identities by utilizing public key identity. Users get comprehensive e-mail signature and encryption, network authentication, and wireless network access because of this.

A Client CA is a CA that we may use to verify a client's identity by storing it in the trusted client CA. We must install and enable appropriate certificates before anyone can use client CA certificates. The system ensures that client CA is not expired and that it is signed by a CA that the system has been configured to recognize when validating it.

5.5 Membership Service Provider

5.5.1 MSP Structure

Membership Service Provider, as the name, suggests it does not provide anything. It is a set of folders that are added to that configuration of the network and used to define an organization. MSP shows who is a network participant or channel member, so by identifying certain rights an actor holds on a node/channel, the MSP converts an identity into a role.

Fabric is a permission network, all the participants of the blockchain network need to prove their identity to the network to transact. Certificate Authorities issues identities by public and private key in the form of key-pair value, as the private key, can never be shared publicly so in that case MSP is required to prove identities.

The ordering service's MSP contains the peer's public key, which is used to verify that the transaction's signature is acceptable. The private key produces a signature on the transaction that only corresponds public key, this is the MSP part that matches both keys. As a result, MSP is the mechanism that allows the rest of the network to trust and recognize that identity without sharing the member's private key. MSP structure looks like,

```
Org1/peer1/localMsp/
|---IssuerPublicKey
|---IssuerRevocationPublicKey
|---cacerts
|       |--   0-0-0-0-8053.pem
|---config.yaml
|---keystore
|       |--   key.pem
|---signcerts
        |--   cert.pem
```

5.5.2 MSP Domain

The private key in the key pair cannot be removed from the channel, so a mechanism is needed to verify the identity of the network member. So, the duties of an MSP are verifying the identities of network members and determining the privileges assigned to a certain member in the network. Therefore, MSPs occur in two different domains in a blockchain network:

Local MSPs, this type of MSP folder contains the individual identity of the network member. Each member of the network should have a local file system. Local MSPs are defined for clients and nodes (peers and orderers) and every node must have a local MSP defined.

Channel MSP, channel MSPs need to determine which members are acting on behalf of which organizations in an application channel. The structure can decide whether a network member has the right to operate on that application channel by checking that member's organization is specific in the channel configuration or not. To do this by specifying every organization and its corresponding MSP/ folder in the configtx.yaml file.

So, Local MSPs are only defined on the file system of the node or user, whereas channel MSP is also instantiated on the file system of every node in the channel and kept synchronized via consensus.

5.6 Smart Contracts

5.6.1 Development

When any given situation is satisfied, a smart contract is an agreement between two parties that enforces certain rules or conditions of discussion. In the computer language, smart contracts are the program logic that is applied to business logic to create an agreement between the engaged entities. Smart contracts may be defining an almost limitless number of commercial use cases involving data integrity in multi-organizational decision-making.

5.6.2 Endorsement

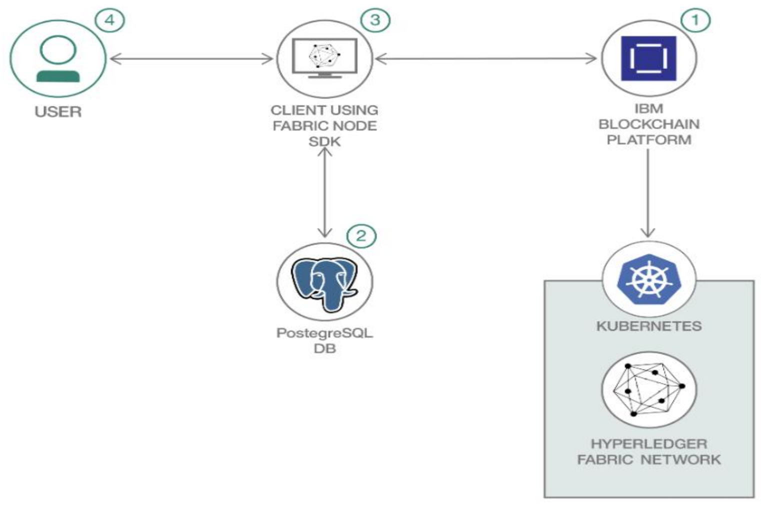
An endorsement policy is associated with each chain code, and it applies to all the contracts established inside it. An endorsement is critical because it specifies which entities in a blockchain network must sign a transaction created by a certain smart contract before it can be considered legal.

5.6.3 Valid Transaction

When a smart contract is activated, it operates on a peer node in the blockchain network that is controlled by an organization. The contract reads and writes the ledger using a set of input parameters called the transaction proposal in connection with its program logic. Changes to the world state are represented by a transaction proposal response (or simply transaction response), which provides a read-write set including both the read states and the new states to be written if the transaction is valid.

5.7 Postgres Wallet

Hyperledger Fabric uses appropriate enrollment certificates while interacting with the blockchain network. The Hyperledger Fabric SDK for Node.js provides APIs to interact with a Hyperledger Fabric blockchain. The Fabric Node SDK provides a default file system wallet for storing Fabric certificates. The file system wallet stores user certificate in folders. This approach does not provide the required security or flexibility, and it also affects scalability. Explore this pattern further to understand how to use a PostgreSQL database as a Fabric wallet.



Role of PostgreSQL

6. Implementation

6.1 CA and MSP

6.2.1 CA Client and Server, MSP

Initially, the GO path needs to be set up correctly by exporting the path, then fabric CA server and client binary tar package is downloaded that will be extracted in the GO path. To implement that 'fabric-ca-server' and 'fabric-ca-client' commands are executed from any current path. Before starting TLS CA container, there is a need of yaml file that needs to be named as docker-compose.yaml. The main 2 factors to be considered in the docker-compose file are Version of the file, the current fabric version. The Fabric-ca version needs to be updated in the image section and in the volumes section, the correct current path needs to be updated.

These factors are going to be repeated before every container is made up. Now, TLS CA container is launched by using the command 'docker-compose up ca-tls'. After executing this command, the TLS CA server starts listening on a secure socket and issuing TLS certificates. Using TLS-CA-CERT.pem file TLS-CA admin is enrolled and peer nodes as peer1-org1, peer2-org1, peer2-org2, peer2-org2, and prderer1-org0 are enrolled. Every organization has its own CA for issuing enrollment certificates, Orderer org0 root CA is started by docker-compose file. Before starting the org0 CA server, the docker-compose file needs to be changed and the folder structure would be according to the volume section in the docker file. Orderer in org0 and admin is registered and enrolled by ca-cert.pem file that is generated in TLS CA folders. After enrolling the identities, the MSP folder for admin will be created in org0 where it consists of fabric-ca-client-config.yaml file, issuer public key, and a few folders to store certificates. The same process is followed for the other two organizations and the MSP structure for all the identities will be created automatically after enrolling and registering.

Likewise, peer1 and peer2 of org1 are enrolled by issuing trusted root certificate if org1 to peer1/assets/ca folder and the certificate file is renamed to org1ca-cert.pem. Changing the name of the file that is present in peer1/assets/tls-msp/keystore folder to key.pem makes it easier while validating. Now admin of org1 is enrolled and the admin certs folder is created automatically in the host peer's MSP folder. After, confirming that the admin certs folder is created in the MSP folder peer’s host machine, then cert.pem file that is present in org1/admin/msp/signcerts folder to admincerts folder in the current peer’s MSP folder.

Finally in the setup process orderer needs to be set up, this process will be almost the same as the before steps. Before, enrolling orderer the trusted root certificate of org0 needs to be copied to org0/orderer/assets/ca/ folder as org0-ca-cert.pem. After enrolling the orderer, the file generated in the keystore of tls-msp of orderer needs to rename to key.pem for future use. Later admin needs to be enrolled so that admin folder will be generated in organization folder i.e., in org0 folder. Admincerts folder needs to be created in org0/orderer/msp/ folder and copy the generated admin certificate to admincerts folder. The cert.pem in the admincerts folder needs to be renamed to orderer-admin-cert.pem for future use. Before orderer is started artifacts need to be created. Genesis Block and Channel Transaction makes artifacts. To create these artifacts, firstly msp folder structure for every organization needs to be created and made accessible easily.

Now by using configtxgen command we generate genesis.block and channel.tx in orderer organization. At this stage, we can start orderer organization by giving the genesis block path to the docker file and a few changes to it as mentioned in step. After the orderer organization is up, CLI containers need to be up the same as by using a docker-compose file with necessary edits.

Once all the containers are up in the docker file, create and join the channel that needs to be happened to do the transaction. Here, we used peer1 to create a channel, this can be achieved by copying the channel.tx file that is generated in orderer organization folder to org1-peer1’s asset folder.

Once the channel is created, then mychannel.block will be created. Here, ‘mychannel’ is the name that we give for our channel while creating and by taking the mychannel.block file in peer1 folder path, remaining peers are joined in the channel.

1. **Set up the blockchain network:** Navigate to the test-network subdirectory within the local clone of the fabric-sample repository.
   cd fabric-sample/test-network
2. **Launch the network:** This next section is required to be the MBSE subdirectory within the same local clone of the fabric-sample repository. Launch the network using startFabaric.sh shell script, this command will spin up a blockchain network comprising peers, orderers, certificate authorities, and more. ./startFabric.sh

- ➢ Initially Fabric server checks for any active networks. If there are any active networks, fabric server downs the network
- ➢ Creates Organization – 1 Identities.
  - ❖ Enrolling the CA Admin
  - ❖ Registering the peer 0
  - ❖ Registering user
  - ❖ Registering the Org admin
  - ❖ Generating the peer 0 msp
  - ❖ Generating the peer-0 tls certificates
  - ❖ Generating the user msp
  - ❖ Generating the org admin msp
- ➢ Creates Organization – 2 Identities
  - ❖ Enrolling the CA Admin
  - ❖ Registering the peer 0
  - ❖ Registering user
  - ❖ Registering the Org admin
  - ❖ Generating the peer 0 msp
  - ❖ Generating the peer-0 tls certificates
  - ❖ Generating the user msp
  - ❖ Generating the org admin msp
- ➢ Creates Orderer Organization Identities
  - ❖ Enrolling the CA Admin
  - ❖ Registering the orderer

- ❖ Registering the orderer Admin
- ❖ Generating the orderer msp
- ❖ Generating the orderer-tls certificates
- ❖ Generating the Admin msp
- ❖ Generating ccp files for Org1 and Org2

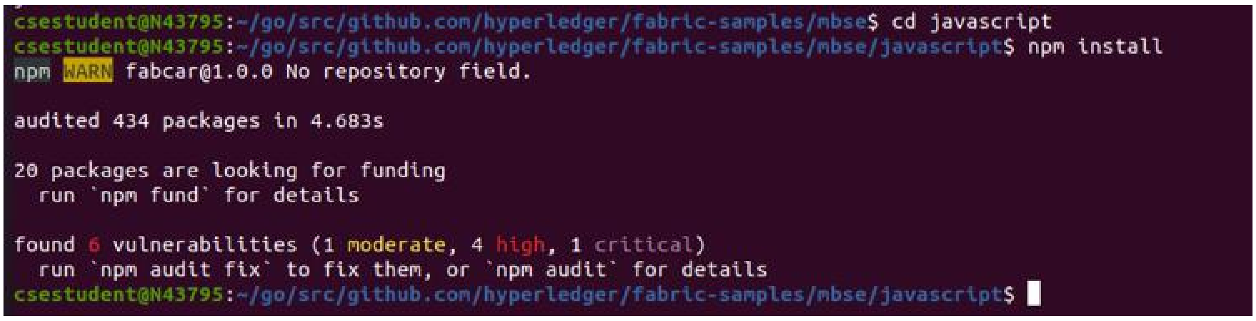
➢ Creates channel – “mychannel”

➢ Chain code will be installed and defined on peer0.org1 and peer0.org2

6.2 Application Execution

Next, we use the MBSE application to interact with the deployed MBSE contract. We start by changing the client “JavaScript” directory to access the smart contracts:

➢ Run npm install command to install all the fabric dependencies for the applications, all the dependencies will be in a JSON file. All the required node modules will also be installed.



```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse$ cd javascript
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ npm install
npm WARN fabcar@1.0.0 No repository field.

audited 434 packages in 4.683s

20 packages are looking for funding
  run `npm fund` for details

found 6 vulnerabilities (1 moderate, 4 high, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ 
```

Enroll Admin: Before registering the user, we need to enroll the admin. After enrolling the admin, we need to load the network configuration and create a new CA Client for interacting with the CA. Then we create in-memory wallet based on the identities. Check for admin in the Postgres database, if admin already exists in the Postgres wallet, system will throw the error that admin is already registered else it will enroll the new admin and updates the certificate and private key into the in-memory wallet and Postgres wallet.

Create a new CA client for interacting with CA

```
const caInfo = ccp.certificateAuthorities['ca.org1.example.com'];
const caTLSCACerts = caInfo.tlsCACerts.pem;
const ca = new FabricCAServices(caInfo.url, { trustedRoots: caTLSCACerts, verify: false },
caInfo.caName);
```

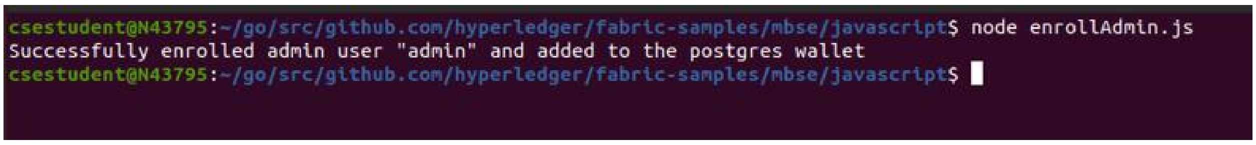
Creating a new in-memory wallet

```
const wallet = await Wallets.newInMemoryWallet();
```

Enrolling the Admin

```
const enrollment = await ca.enroll({ enrollmentID: 'admin', enrollmentSecret:
'adminpw' });
const x509Identity = {
        credentials: {
                certificate: enrollment.certificate,
                privateKey: enrollment.key.toBytes(),
        },
        mspId: usermspId,
        type: 'X.509',
};
```

As shown in the above code snippet, admin's certificate and private key are stored in the credentials and assigned to x509Identity. Then we update the certificate and private key into Postgres wallet.



```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node enrollAdmin.js
Successfully enrolled admin user "admin" and added to the postgres wallet
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$
```

So, to enroll the admin, we need to execute the command "node enrollAdmin.js". Then admin will be enrolled successfully and added to the Postgres wallet as displayed in the above screenshot.

Register User: User will be registered in the front end / UI. For example, here we register two users. One user for Organization 1 and another user for organization 2. While registering the user, it checks if admin is enrolled or not. User can register only if admin is enrolled in the system. At the time of user registration only, Front End/UI interacts with the Hyper Ledger Fabric to register the user and stores the certificate and private key into the Postgres wallet and build the user objects for the authentication with certificate authority. Then it updates the same to the in-memory wallet and Postgres wallet.

Building User Object for authentication with CA and MSP

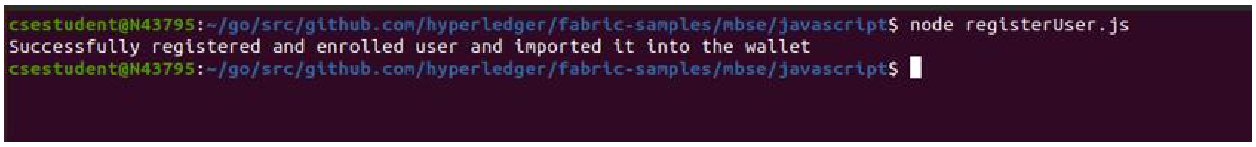
```
const adminIdentity = await wallet.get('admin');
const provider =
wallet.getProviderRegistry().getProvider(adminIdentity.type);
```

Registering User

```
const secret = await ca.register({
affiliation: 'org1.department1',
enrollmentID: user,
role: 'client'
}, adminUser);
```

As shown in the above code snippet, we are registering the user for Organization -1 and storing in “secret”. Similarly, we register user for the organization – 2 as well. From the CLI, we can register the user by executing the command “node registerUser.js”.

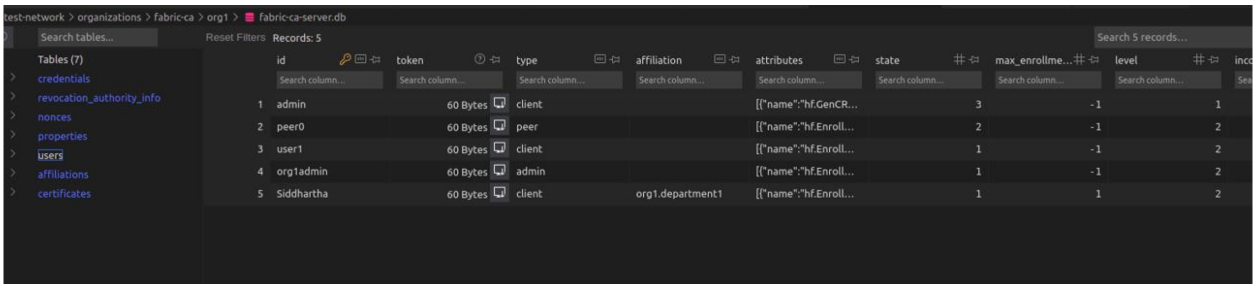
For Organization – 1:



```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node registerUser.js
Successfully registered and enrolled user and imported it into the wallet
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$
```

So, after executing the command, it says successfully registered and enrolled user and imports into the Postgres wallet. This is for the user belonging to the Organization – 1. We can check the confirmation of the user in the “fabric-ca-server database” by viewing its affiliation and id.

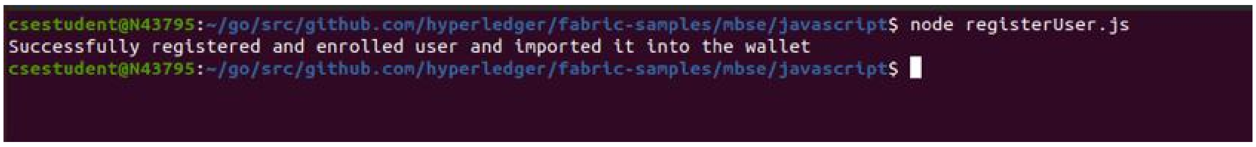
Here is the screenshot below:



	id	token	type	affiliation	attributes	state	max_enrollme...	level
1	admin	60 Bytes	client		[{"name":"hf.GenCR...	3	-1	1
2	peer0	60 Bytes	peer		[{"name":"hf.Enroll...	2	-1	2
3	user1	60 Bytes	client		[{"name":"hf.Enroll...	1	-1	2
4	org1admin	60 Bytes	admin		[{"name":"hf.Enroll...	1	-1	2
5	Siddhartha	60 Bytes	client	org1.department1	[{"name":"hf.Enroll...	1	1	2

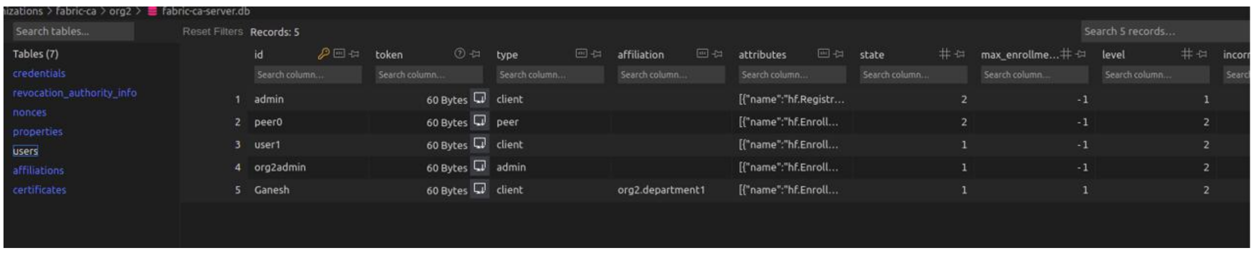
As shown in the above screenshot – in “id” = 5 there is a user registered with name = “Siddhartha” and the affiliation is Organization – 1. Similarly, now we will be registering the user – 2.

For Organization – 2:



```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node registerUser.js
Successfully registered and enrolled user and imported it into the wallet
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ 
```

Now we registered another user for Organization – 2. As shown in the below screenshot, "id" = Ganesh and the affiliation is Organization – 2.



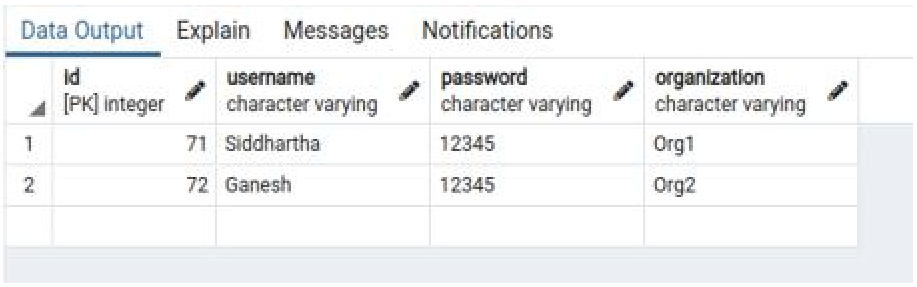
izations > fabric-ca > org2 > fabric-ca-server.db

Tables (7): credentials, revocation_authority_info, nonces, properties, users, affiliations, certificates

Records: 5

	id	token	type	affiliation	attributes	state	max_enrollme...	level	incor
1	admin	60 Bytes	client		[{"name":"hf.Registr...	2	-1	1	
2	peer0	60 Bytes	peer		[{"name":"hf.Enroll...	2	-1	2	
3	user1	60 Bytes	client		[{"name":"hf.Enroll...	1	-1	2	
4	org2admin	60 Bytes	admin		[{"name":"hf.Enroll...	1	-1	2	
5	Ganesh	60 Bytes	client	org2.department1	[{"name":"hf.Enroll...	1	1	2	

Next, we can check the Postgres database if the user is stored or not in the below screenshot.



	id [PK] integer	username character varying	password character varying	organization character varying
1	71	Siddhartha	12345	Org1
2	72	Ganesh	12345	Org2

As shown, the user is stored with its "id", "username", "password" and "Organization".

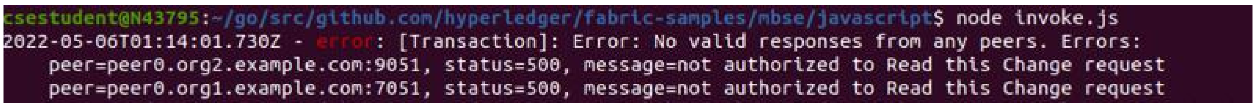
Invoking a Smart Contract and Updating Identity: Once a user is registered, he executes some transactions. So, in invoke.js initially it loads the network configuration and creates a new in-memory based wallet for managing identities. It checks if the user exists or not in the Postgres wallet, and if exists, it retrieves the users' certificate and private key into in-memory wallet and authenticates them before the user performs any transaction. If the user is not present, system throws an error that "'First register the user before retrying here". Then we create a new gateway for connecting to our peer node where we get the network to deploy our smart contract. Then we get the smart contract from the network and Transact the Asset Action and submit the specified transaction. At the end the network will be disconnecting from the gateway.

Gateway Connection for connecting to our peer node

```
const assetActionPath = path.resolve(__dirname,
'AssetAction_C_CR.json');
// const assetActionPath = path.resolve(__dirname,
const gateway = new Gateway();
await gateway.connect(ccp, { wallet, identity: userId, discovery: { enabled: true, asLocalhost:
true } });
```

Transacting the Asset Action

So as per the above code snippet, the user is trying to perform the Create Asset Action. User will not be able to perform the specified transaction because he is not authorized to do. The user with the role CSE will be able to perform the specified Asset Action.



```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node invoke.js
2022-05-06T01:14:01.730Z - error: [Transaction]: Error: No valid responses from any peers. Errors:
    peer=peer0.org2.example.com:9051, status=500, message=not authorized to Read this Change request
    peer=peer0.org1.example.com:7051, status=500, message=not authorized to Read this Change request
```

```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node invoke.js
2022-05-06T01:14:01.730Z - error: [Transaction]: Error: No valid responses from any peers. Errors:
    peer=peer0.org2.example.com:9051, status=500, message=not authorized to Read this Change request
    peer=peer0.org1.example.com:7051, status=500, message=not authorized to Read this Change request
```

As displayed in the above screenshot, User is not authorized to perform the Create Change Request. So, in order to perform the transaction by the user, we need to update the user and assign the role CSE. Then user will be able to perform the Transaction.

Update Identity: Like the above process, here initially we create a new CA client for interacting with CA and new in-memory based wallet to manage identities. It checks if the user exists or not in the Postgres wallet, and if exists it retrieves the users certificate and private key into in-memory wallet and authenticate them before the user performs any transaction. If the user is not present, system throws a error that "'First register the user before retrying here". It builds a user object for authentication with Certificate Authority and MSP. Then it updates the identity using the update identity class. Then it re-enrolls the user to get a updated certificate. At the end, in-memory wallet and Postgres wallet will be updated.

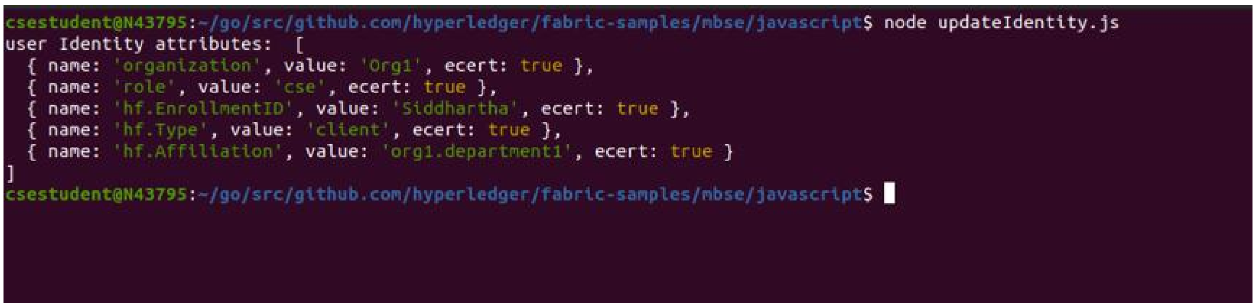
Using the Identity Service class to update the user Identity

```
const identityService = ca.newIdentityService();
//Passing the id passed at registerUserAPI
const response = await identityService.update(userName, updateObj,
adminUs
```

```
// update identity
let updateObj = {
        type:"client",
        affiliation:"org1.department1" ,
        attrs: [{ name: "organization", value: "org1", ecert:
true },
        { name: "role", value: "cse", ecert: true }] ,
        caname:"ca_peerOrg1"
}
```

As shown in the above code snippet, we are assigning the role CSE for the User because CSE has authorities to perform the transaction. So now after updating the user, he will be able to submit / perform the transaction.

After executing Update Identity, the result will be like:

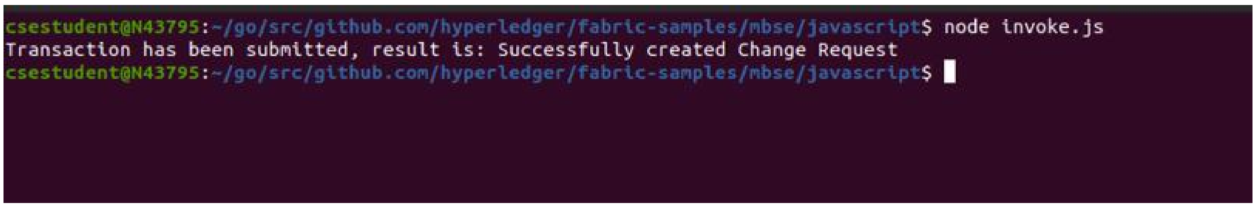


```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node updateIdentity.js
user Identity attributes:  [
  { name: 'organization', value: 'Org1', ecert: true },
  { name: 'role', value: 'cse', ecert: true },
  { name: 'hf.EnrollmentID', value: 'Siddhartha', ecert: true },
  { name: 'hf.Type', value: 'client', ecert: true },
  { name: 'hf.Affiliation', value: 'org1.department1', ecert: true }
]
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$
```

Submit the specified transaction

```
const result = await contract.submitTransaction('ManageMBSEAssets',
JSON.stringify(assetActionJson));
// await contract.submitTransaction('CreateCR', JSON.stringify(assetActionJson));
// await contract.submitTransaction('UpdateCR', JSON.stringify(assetActionJson));
// await contract.submitTransaction('GetIdentityAttribute', 'desg');
```

As per the above code snippet, user is submitting the Contract. He will be able to submit now because he was assigned as “CSE”.



```
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$ node invoke.js
Transaction has been submitted, result is: Successfully created Change Request
csestudent@N43795:~/go/src/github.com/hyperledger/fabric-samples/mbse/javascript$
```

So, as shown above, after executing invoke.js, the user will be able to successfully create the Change Request. Change Request is the contract the user invoked. Now it is not authorized for the other user who belongs to the Organization – 2 even though he is “CSE”. This transaction is authorized only for the user with role CSE belongs to the Organization – 1.

6.3 Smart Contracts

Ideology for implementation of ABAC in Smart Contract

In this project, we specified security and privacy policies based on Attribute-Based Access Control and implemented them in Hyperledger fabric using smart contracts. The ideology is to implement these security and privacy policies by following:

(a) Understanding ABAC Project

(b) Suggested an idea to write attributes in JSON and use them in user creation

(c) Developed java scripts to create users, reading attributes from JSON, and developed java scripts-based user scripts to automate the process and facilitate feature enhancements

(d) Created the project structure to represent the model with a few attributes

(e) Implemented smart contracts in chain code considering the policies

Smart Contract for Creating a Change Request

```
func (s *SmartContract) CreateCR(ctx contractapi.TransactionContextInterface, assetAction
AssetAction) (string, error) {
isAuthorized := s.CreateCR_SP(ctx, assetAction.BCAsset.CRDecision,
assetAction.BCAsset.Project.OrgRoles)
    if !isAuthorized {
        return "", fmt.Errorf("not authorized to create a Change request")
    }
    exists, err := s.AssetExists(ctx, assetAction.BCAsset.BCAssetId)
    if err != nil {
        return "Error ", fmt.Errorf("could not read from world state. %s", err)
    } else if exists {
        return "Error ", fmt.Errorf("the asset %s already exists",
assetAction.BCAsset.BCAssetId)
    }

    bytes, _ := json.Marshal(assetAction.BCAsset)

    return "Successfully created Change Request",
ctx.GetStub().PutState(assetAction.BCAsset.BCAssetId, bytes)
}
```

Description: The input for the smart contract is USAA (user-specified asset action), the output is a statement that gets displayed after a successful transaction or it displays error if smart contract transaction is failed. Initially, we are getting the invoker of SC, assetAction is the type of action that must be performed on the asset which is passed as an argument. Next, the create CR policy is checked. It states that the CRDecision must be empty, and the user role must be CISE belonging to a lead organization. Next, the user is verified for the corresponding privilege. The data passed as an argument is verified against the BCAsset data. If the Security policy is successfully executed, we are checking if the asset already exists in the blockchain or not. If yes, then an error is thrown else a Change Request document is created.

Smart Contract for updating the Smart Contract

```
func (s *SmartContract) UpdateCR(ctx contractapi.TransactionContextInterface, assetAction
AssetAction) (string, error) {
    isAuthorized := s.UpdateCR_SP(ctx, assetAction.AttributesToUpdate.CRDecision,
assetAction.BCAsset.Project.OrgRoles)
    if !isAuthorized {
        return "Error ", fmt.Errorf("not authorized to update a Change request")
    }
    asset, err := s.ReadCR(ctx, assetAction)
    if err != nil {
        return "", err
    }
    bcAsset := new(BCAsset)
    err = json.Unmarshal([]byte(asset), bcAsset)
    if err != nil {
        return "Error ", fmt.Errorf("error. %s", err)
    }
    bcAsset.CRSubmissionTime = assetAction.AttributesToUpdate.CRSubmissionTime
    bcAsset.CRDecision = assetAction.AttributesToUpdate.CRDecision
    bytes, _ := json.Marshal(bcAsset)
    return "Successfully Updated CR Decision ", ctx.GetStub().PutState(bcAsset.BCAssetId,
bytes)
}
```

Description: The input for the smart contract is USAA (user-specified asset action), the output is a statement that gets displayed after a successful transaction or it displays error if smart contract transaction is failed. Initially, we are getting the invoker of SC, assetAction is the type of action that must be performed on the asset which is passed as an argument. Here, the action is update. Next, the update CR policy is checked. It states that the CRDecision must be empty initially and the user role must be CSE belonging to a lead organization. Next, the user is verified for the corresponding privilege. The data passed as an argument is verified against the BCAsset data. If the Security policy is successfully executed, we are checking if the asset exists in the blockchain or not. If yes, then an error is thrown else an update transaction takes place.

```
func (s *SmartContract) ReadCR(ctx contractapi.TransactionContextInterface, assetAction
AssetAction) (string, error) {
    isAuthorized := s.ReadCR_SP(ctx, assetAction.BCAsset.Project.OrgRoles)
    if !isAuthorized {
        return "Error ", fmt.Errorf("not authorized to Read this Change request")
    }
    assetActionJSON, err := ctx.GetStub().GetState(assetAction.BCAsset.BCAssetId)
    if err != nil {
        return "Error ", fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetActionJSON == nil {
        return "Error ", fmt.Errorf("the asset %s does not exist",
assetAction.BCAsset.BCAssetId)
    }
    // return &bcAsset, nil
    return string(assetActionJSON[:]), nil
}
```

Smart Contract for reading a Change Request

Description: The input for the smart contract is USAA (user-specified asset action), the output is a statement that gets displayed after a successful transaction or it displays error if Smart Contract transaction is failed. Initially, we are getting the invoker of SC, assetAction is the type of action that must be performed on the asset which is passed as an argument. Here, the action is read. Next, the read CR policy is checked. It states the user role must be CSE belonging to a lead organization. Next, the user is verified for the corresponding privilege. The data passed as an argument is verified against the BCAsset data. If the Security policy is successfully executed, we are checking if the asset exists in the blockchain or not. If yes, then an error is thrown else an update transaction takes place.

6.3.1 Implementation of security and privacy policies in smart contracts

(a) Create Change Request: The Policy is that only the CISE from the lead organization of a project can create a new change request document of the project.

Target: Change Request document which inside the blockchain

Action: Create

Combining Algorithm: denyOverrides

Conditions: The user works as the lead organization of project, the user works on the current project and lastly, the User role must be CISE (Chief Integration System Engineer), CRDecisions must be empty initially.

```
func (s *SmartContract) CreateCR_SP(ctx contractapi.TransactionContextInterface, crDecision
CRDecisionStruct, orgRoles []OrgRolesStruct) bool {
	fmt.Println("Executing CreateCR Policy!!!")
	userOrganization := s.GetIdentityAttribute(ctx, "organization")
	userRole := s.GetIdentityAttribute(ctx, "role")
	return (crDecision == CRDecisionStruct{} && s.Contains(s.GetOrgId(orgRoles, "lead"),
userOrganization) && userRole == "cse")
}
```

(b) **Update Change Request:** The policy verifies whether the request is authorized to update a change request or not.

Target: Change Request document which is inside the blockchain

Action: Update

Combining Algorithm: denyOverrides

Conditions: The user works as the lead organization of project, the user works on the current project and lastly, the User role must be CSE (Chief Integration System Engineer), CRDecisions must be empty initially.

```
func (s *SmartContract) UpdateCR_SP(ctx contractapi.TransactionContextInterface, crDecision
CRDecisionStruct, orgRoles []OrgRolesStruct) bool {
    fmt.Println("Executing UpdateCR Policy!!!")

    userOrganization := s.GetIdentityAttribute(ctx, "organization")
    userRole := s.GetIdentityAttribute(ctx, "role")
    return (crDecision != CRDecisionStruct{} && s.Contains(s.GetOrgId(orgRoles, "lead"),
userOrganization) && userRole == "cse")
}
```

(c) Insert a Change Request Decision: The policy verifies that the request is authorized to add a CR decision into a change request object or not.

Target: Inserting a CR Decision into a Change request object

Action: Insert

Combining Algorithm: denyOverrides

Conditions: The user organization is control board, and he must be a manager. “IsWithdrawn” is the attributes to update.

```
func (s *SmartContract) InsertACRDecision_SP(ctx contractapi.TransactionContextInterface,
attributesToUpdate []string, orgRoles []OrgRolesStruct) bool {
	fmt.Println("Executing InsertACRDecision Policy!!!")
	userOrganization := s.GetIdentityAttribute(ctx, "organization")
	userRole := s.GetIdentityAttribute(ctx, "role")
	return (s.Contains(attributesToUpdate, "IsWithdrawn") && s.Contains(s.GetOrgId(orgRoles,
"controlBoard"), userOrganization) && userRole == "manager")
}
```

(d) **Withdraw a Change Request Decision:** The policy is that the CSE of a lead organization can withdraw a Change request.

Target: Withdrawing a Change request.

Action: Withdraw

Combining Algorithm: denyOverrides

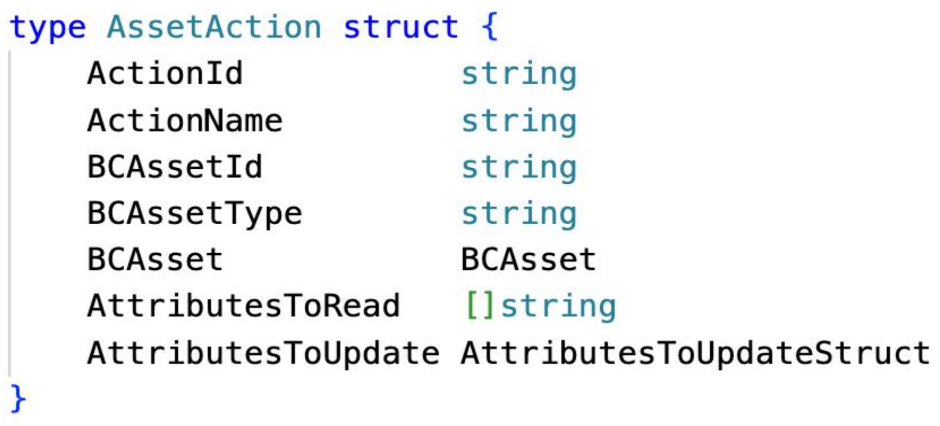
Conditions: The user organization is lead and he must be a CSE. IsWithdrawn and WithdrawnTime are the attributes to read.

```
func (s *SmartContract) WithdrawACR_SP(ctx contractapi.TransactionContextInterface,
attributesToRead []string, orgRoles []OrgRolesStruct) bool {
    fmt.Println("Executing WithdrawACR Policy!!!")
    // user.organization == action.newobject.project.leadOrganization
    // user.abac.bc_orgrole == 'cse'
    // (action.newobject.attributesToUpdate == "IsWithdrawn" &&
action.newobject.attributesToUpdate == "WithdrawnTime")
    userOrganization := s.GetIdentityAttribute(ctx, "organization")
    userRole := s.GetIdentityAttribute(ctx, "role")
    return (s.Contains(attributesToRead, "IsWithdrawn") && s.Contains(attributesToRead,
"WithdrawnTime") && s.Contains(s.GetOrgId(orgRoles, "lead"), userOrganization) && userRole ==
"cse") }
```

7. MBSE Struct

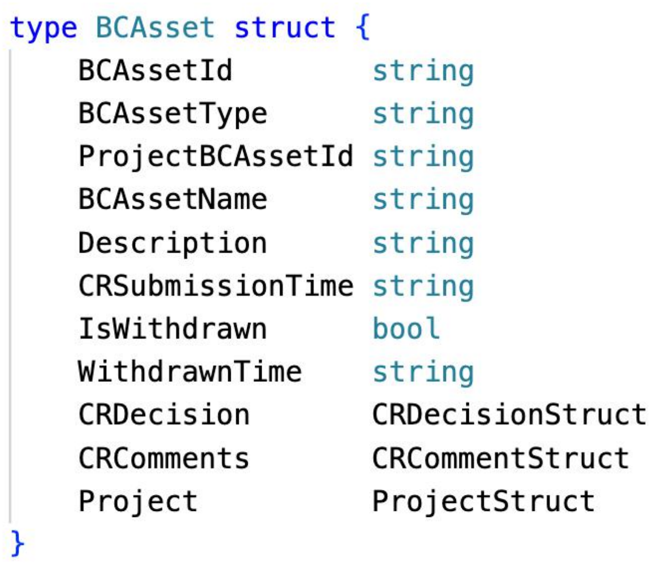
It will contain various methods to do basic create, read, and update operations in the blockchain. The smart contract will get the JSON string as an input and will be unmarshalled to struct in GOLANG to do the validation operations and it will be stored in the blockchain in byte data.

7.1 The Golang struct for AssetAction:



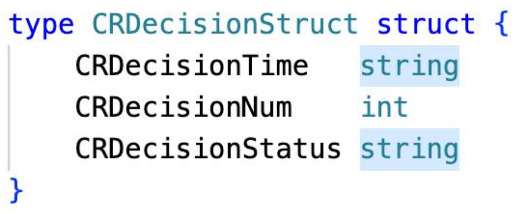
```go
type AssetAction struct {
    ActionId           string
    ActionName         string
    BCAssetId          string
    BCAssetType        string
    BCAsset            BCAsset
    AttributesToRead   []string
    AttributesToUpdate AttributesToUpdateStruct
}
```

7.2 The Golang struct for BCAsset:



```go
type BCAsset struct {
    BCAssetId        string
    BCAssetType      string
    ProjectBCAssetId string
    BCAssetName      string
    Description      string
    CRSubmissionTime string
    IsWithdrawn      bool
    WithdrawnTime    string
    CRDecision       CRDecisionStruct
    CRComments       CRCommentStruct
    Project          ProjectStruct
}
```

7.3 The Golang struct for CRDecision:



```go
type CRDecisionStruct struct {
    CRDecisionTime   string
    CRDecisionNum    int
    CRDecisionStatus string
}
```

8. Requirements covered in the project

- Understanding and learning all technical terms of the project
- Learning the basics of Blockchain and Fabric
- Installing Hyperledger Fabric latest version and its prerequisites
- Setting up the initial fabric test network
- Understanding the code to develop SC/CC effectively
- Understanding the MBSE asset model deep enough for development
- Understanding the MBSE Gateway project sufficiently to capture its ABAC attributes
- Capturing most of the selected Gateway MBSE project requirements
- Developing an accurate and complete set of security policies and the required ABAC attributes for the Gateway MBSE project
- Implementing the security policy portion of the SC/CC
- Implementing ChangeRequest policy, DocumentPackage policy and BCDocument policy in SC/CC using GOLANG
- Implementing CRUPD operations on assets in Smart Contract
- Registering and enrolling identities with CA by adding user attributes, storing them in the in-memory wallet
- Implemented Angular application for user registration
- Storing the users certificate and private key into the Postgres Database
- Retrieving the users certificate from the Postgres Database to the in-memory wallet
- Build the user object authentication with CA

9. Individual Contributions

Individual teammate contribution to the Security and privacy modeling, Implementation with XACML/ALFA and Fabric

Venkata Satya Siddhartha Illa:

Installed Hyperledger Fabric and its prerequisites

Worked on implementing the Security and privacy policies

Enrolled and Registered user identities

Worked on Fabric CA and MSP configuration

Implemented Smart Contracts based on the SC algorithm

Implemented helper functions for executing security policies

Prepared the readme file for the project and uploaded to Github

Integrated the project

Venkata Naga Bhaavagni Maddi:

Installed Hyperledger Fabric and its prerequisites

Worked on Sample Book Application

Implemented Smart Contracts based on the SC algorithm

Performed Manual testing to debug errors

Worked on minutes of meeting reports

Worked on Final report document

Farhana Begum Shaik:

Installed Hyperledger Fabric and its prerequisites

Worked on Sample Book Application

Implemented Smart Contracts based on the SC algorithm

Worked on Design Document

Worked on Final presentation

Sripada Vallabh Kaparthi:

Installed Hyperledger Fabric and its prerequisites

Worked on UI and Postgres Database

Stored User’s Certificates and Private key in Postgres Database with user authentication

Retrieving the certificates from Postgres database

Worked on identifying Attributes from BPMN

Performed Manual testing

Worked on Requirement Document and Final Report

Developed and handled Team Website

Scheduled Mentor Meetings and Email Communication with professor

Ganesh Nyaupane:

Installed Hyperledger Fabric and its prerequisites

Worked on enrolling and registering user identities using In-memory wallet and Postgres wallet

Worked on retrieving and updating identity attributes

Supported for integration of the project

Worked on Final presentation

Worked on weekly report

10. Future Work

- Need to manage more ABAC attributes while registering the users in CA and MSP
- Need to store relevant ABAC attributes in the Postgres Database for the users
- Integration of the Project-1 UI application with Security and privacy modeling application to enroll Admin, Users and invoking the smart contracts with the Security policies implemented
- We can use dynamic struct with the use of maps in order to avoid storing null values in the world state
- Complete Database Schema has to be developed

11. Conclusion

Role-based access control (RBAC) and attribute-based access control (ABAC) are one of the two popular access control mechanisms for controlling the authorization of users. The primary difference between RBAC and ABAC is that RBAC provides access to resources based on user roles, While ABAC provides access rights based on user's subject, action, environment, and resource attributes, which allows for higher levels of access control. In this project, the BPMN workflow defines many roles and attributes which makes implementing ABAC methodology a right choice. By analyzing the BPMN workflow, we have identified the relevant ABAC attributes and the business workflow. We have used ALFA to specify the security and privacy requirements in the form of security policies, rules and conditions. While registering the users we added relevant ABAC attributes to those identities. Also, we have added some of the action attributes that can take place on a AssetAction. We have implemented the security policies using smart contracts which can be easily accessed by the Asset transaction smart contracts. These security policies were expected to authorize the transaction based on the ABAC attributes of the user and the BCAsset. Whenever, a user invokes a transaction, the respective security policy enforces the access control by reading the client identities, ABAC attributes and BCAsset attributes by evaluating against the conditions that are defined in the security policy. We have implemented policies for ChangeRequest, BCDocument and DocumentPackage. Apart from this, we have replaced the file system wallet with a Postgres database and the in-memory wallet for better management of ABAC attributes and security. In future, we need to manage some of the ABAC attributes in the Postgres database. Thus, management of attributes will be easier. Therefore, we enforced an additional layer of security and access control by implementing the ABAC methodology.

12. References

[1] https://hyperledger-fabric-ca.readthedocs.io/en/latest/

[2] https://github.com/IBM/fabric-postgres-wallet

[3] https://github.com/hyperledger/fabric-samples

[4] https://alfa-lang.io/intro.html

[5] https://hyperledger.github.io/fabric-sdk-node/

[6] https://www.json.org/json-en.html

[7] https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-abac/README.md

[8] https://hyperledger-fabric.readthedocs.io/en/release-2.4/whatis.html

[9] https://github.com/IBM/fabric-postgres-wallet