A Decentralized KYC Verification Process for Banks

A Decentralised KYC Verification Process for Banks

Origin of KYC

Know your customer or KYC originated as a standard to fight against the laundering of illicit money flowing from terrorism, organised crime and drug trafficking. The main process behind KYC is that government and enterprises need to track the customers for illegal and money laundering activities. Moreover, KYC also enables banks to better understand their customers and their financial dealings. This helps them manage their risks and make better decisions.

Need for KYC

Taking in from the origin of KYC, we can state that there are four major sectors in banking where KYC is needed. They are as follows:

- **Customer Admittance:** This sector defines making anonymous accounts as restricted entry into the banking system. In other words, no anonymous accounts are allowed. Preliminary information, such as names, date of birth, addresses and contact numbers, is to be collected to provide banking service.
- **Customer Identification:** In the case of suspicious banking transactions through a customer, customer accounts can be tracked and flagged. Further, they can be sent for processing under the bank head office for review.
- **Monitoring of Bank Activities:** Suspicious and doubtful activities in any account can be zeroed in by the bank after understanding its customer base using KYC.

- **Risk Management:** Now that the bank has all the preliminary information and the activity pattern, it can assess the risk and the likelihood of the customer being involved in illegal transactions.

These requirements make the KYC process an essential entity in the banking and financial world. The traditional KYC process is already in place in some banks, but there are major challenges related to the process, and through this case study, we will assess and tackle these challenges. Let’s first list the challenges related to the traditional KYC process.

Problems/Challenges in KYC

- The disparity in specifications for KYC
    - Every bank has its own KYC process set up, and customers need to do the KYC again and again for each bank.
    - Due to the lack of KYC standards, compiling each request is time-consuming.
- Adverse impact on customer relationship
    - It becomes irksome for customers to provide the same information to different banking entities and industries.
    - Banks sometimes even follow up with customers to get more details for KYC.
- Escalating costs and time for banks
    - A recent study concluded that overheads of KYC in a bank increase the onboarding cost for a customer by 18% and the minimum time required to 26 days.

Solution Using Blockchain

The blockchain is an immutable distributed ledger shared with everyone involved in the network. Every participant interacts with the blockchain using a publicprivate cryptographic key combination. Moreover, immutable record storage is provided, which is extremely hard to tamper with.

Banks can utilise the feature set of blockchain to reduce the difficulties faced by the traditional KYC process. A distributed ledger can be set up between all the banks, where one bank can upload the KYC of a customer and other banks can vote on the legitimacy of the customer details. KYC for the customers will be immutably stored on the blockchain and will be accessible to all the banks in the blockchain.

This case study is divided into three parts to achieve the solution. They are as follows:

Phase 1:

- Banks add customers and their data on the network.
- Whenever any new data is needed to be appended, the ledger could enable encrypted updates of the data. Mining will make sure that the data gets confirmed over the blockchain and is distributed to all other banks.
- Banks can modify the data of the customers present in the database. In phase 1, any bank can modify the data of the customer. In phase 2, we will add admin and bank functionalities, which will provide the necessary restrictions over the network and data of the customers.
- Banks can also view customer data.

Phase 2:

- Admin functionalities are provided for the system, where an admin can track the actions such as upload or approval of KYC documents performed by banks.
- The admin can block any bank from doing a KYC; the admin can also add new banks or remove untrusted banks from the smart contract.

- Whenever a new customer enters into the system, a bank initiates a KYC request for the customer with the additional data provided by the customer to the bank. Any bank can raise the KYC request.
- Once a KYC request of a customer is added, any bank can upvote or downvote, stating their stand on the data provided by the customer.
- The bank can remove the KYC request of the customer.
- Now, the customer struct will also store the number of upvotes and downvotes, and the KYC status of the customer. For the KYC status to be true for any customer, the number of upvotes should be greater than the number of downvotes. Also, if one-third of the total number of banks downvote the customer, then the KYC status is set to false even if the number of upvotes is greater than that of downvotes for that customer.
- Customers' KYC status will be stored on the chain depending on the number of upvotes/downvotes.
- Banks also report the other banks to make sure that the banks are secure and not tampered with for the KYC process. This identifies whether the bank is corrupted and whether it is uploading fake KYC. This rating will help us to judge the bank activities and remove the fraudulent bank from our network.
- The admin can anytime disallow the bank from upvoting/downvoting.
- Depending on the number of reports and the number of banks present in the network, it will be decided whether any bank is allowed to downvote or upvote. If any bank gets reported more than one-third of the banks present in the network, it will not be allowed to do KYC anymore.
- You can use some other logic to identify corrupted banks over the network. (For example, if more than half of the banks report the bank or the upvoted customers by a bank get more than a threshold number of downvotes by the other). If a bank is corrupted, set the '**isAllowedToVote**' variable of the bank struct to false.

Phase 3:

- In this phase, the smart contract will be deployed over a private network that is put up between various banks.
- Banks can use the functionalities of the smart contract over this private Ethereum network.
- Banks need to have an account on the private network to interact with the smart contract.

Important:
Phase 1 of this case study is ungraded and meant for your practice only. Phase 2 and phase 3 of this project are graded, and you shall be evaluated on phase 2 and phase 3 only.

Creation of KYC Upload Process over Blockchain

Details of Phase 1:

Bank Accounts and Bank Functionalities

There are two types of accounts that can exist in your Ethereum network: Admin and banks. There will be only one admin account and multiple bank accounts. Only the admin can add or remove a bank. Admin functionalities include adding/removing banks, which will be covered in phase 2. In this phase, we will discuss only the banks and the functions that they can call. Banks have the following functionalities:

- To add a customer to the customer list
- To modify customer details. Any bank can use this function for now in phase 1

Bank and Customer Details:

A **customer** is stored as a struct type. The customer struct needs to have the following components:

- **Username of the customer -** Username is provided by the customer and is used to track the customer details. Here, for the ease of coding, we have assumed that the username is unique and is the primary identifier of a record in the customer list.
    - **Datatype -** string
- **Customer data -** This is the customer's data or identity documents provided by the customer. Customer data is the KYC data of the customer.
    - **Datatype -** string
- **Bank -** This is a unique address of the bank that validated the customer account.
    - **Datatype -** address

Similarly, you need to create a **bank** struct. The bank struct will have the following components:

- **Name -** This variable specifies the name of the bank/organisation.
    - **Datatype -** string
- **ethAddress -** This variable specifies the unique Ethereum address of the bank/organisation.
    - **Datatype -** address
- **regNumber -** This is the registration number for the bank.
    - **Datatype -** string

Functions for Banks in Your Smart Contract

You need to write the following functions in your smart contract. These are the same functions as those mentioned above.

- **Add Customer -** This function will add a customer to the customer list.
    - **Parameter** - Customer name and data as string type

- **Modify Customer** - This function allows a bank to modify a customer's data. For phase 1, any bank can modify the data.
    - **Parameters** - Customer name as string, a new hash of the new customer data as string type
- **View Customer** - This function allows a bank to view the details of a customer.
    - **Parameter** - Customer name as string type
    - **Return** - This function will return the customer data in string format.

Note: After performing phase 1, you will move to phase 2, which is the graded portion of this case study. There, you will define the voting functionality by other banks over the customer data. Also, admin functionalities will be defined along with restrictions on who is allowed to vote on KYC.

In phase 2, the bank will raise a KYC request of a customer with the necessary data. Now, any bank in the network can verify the data and upvote or downvote as per its assessment.

The coding standards and rubrics for grading are stated in the next session.

Points to keep in mind while writing the smart contract:

- You need to add certain checks from your side. For example, when adding a customer to the customer list, make sure that the customer is not already present in the customer list. If the customer is already present, then reject the request.
- You can use bytes32 instead of a string. Similarly, you can use mappings to represent your lists such as the requests list and the customer list. Bytes32 is recommended because it uses lesser gas than the string data type.
- Try to avoid any loops in the smart contract.
- Make use of comments in solidity to make the code easy to understand.

- For phase 1, anyone can use functions such as add customer. In phase 2, the admin will add/remove the banks and also decide whether any bank is allowed to upvote/downvote.

Creation and Verification of KYC over Blockchain

Phase 2 details:

For this phase, you need to add a KYC request struct, and edit the customer and bank structs as follows:

Customer:

- **Username of the customer** - Username is provided by the customer and is used to track the customer details. To ease coding for you, you can assume that this is unique for a customer.
    - **Datatype -** string
- **Customer data** - This is the customer's data or identity documents provided by the customer. This is unique for a customer.
    - **Datatype -** string
- **kycStatus** - This is the status of the KYC request. If the number of upvotes/downvotes meet the required conditions, set kycStatus to true; otherwise, set it to false.
    - **Datatype -** boolean

- **Downvotes** - This is the number of downvotes received from other banks over the customer data.
    - **Datatype -** unsigned integer
- **Upvotes** - This is the number of upvotes received from other banks over the customer data.
    - **Datatype -** unsigned integer
- **Bank** - This is a unique address of the bank that validated the customer account.
    - **Datatype -** address

Banks (Organisation):

- **Name -** This variable specifies the name of the bank/organisation.
    - **Datatype -** string
- **ethAddress -** This variable specifies the unique Ethereum address of the bank/organisation.
    - **Datatype -** address
- **complaintsReported -** This is the number of complaints against this bank done by other banks in the network.
    - **Datatype -** unsigned integer
- **KYC_count -** This is the number of KYC requests initiated by the bank/organisation.
    - **Datatype -** unsigned integer
- **isAllowedToVote -** This is a boolean to hold the status of the bank. If set to false, the bank cannot upvote/downvote any more customers.
- **regNumber -** This is the registration number for the bank. This is unique.
    - **Datatype -** string

KYC Request:

The Smart Contract will contain a KYC request from the bank, which will be initiated for a customer. This request will have the following data associated with it:

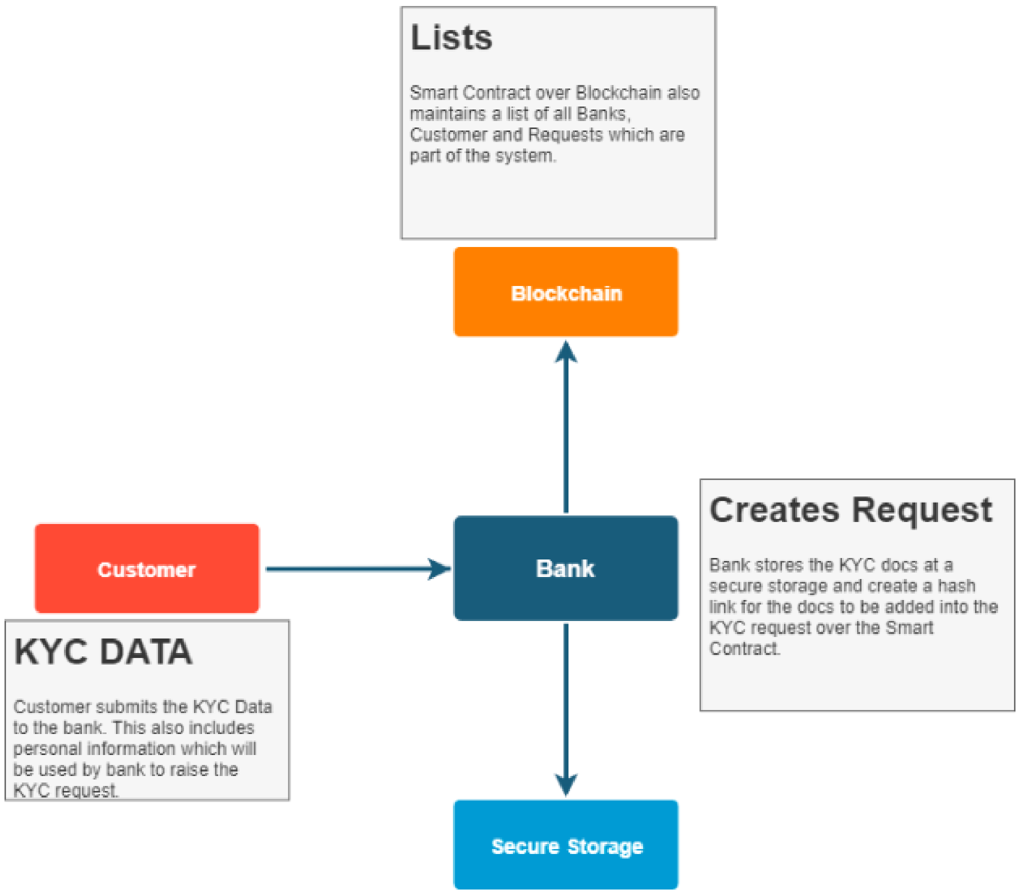
- **Username -** Username will be used to map the KYC request to the customer data. A person has a unique Username.
    - **Datatype -** string
- **Bank address -** Here, Bank address is a unique account address for the bank, which can be used to track the bank.
    - **Datatype -** address
- **Customer data -** This is the customer's data or identity documents provided by the customer. This is unique for each request.
    - **Datatype -** string

Use Case

If the user accepts to share data, the following steps will take place:

- A bank will check the blockchain to fetch the hash of the customer data and use the hash to fetch the actual customer data from a secure storage.
- Any bank can raise the KYC request of the customer given that the customer has provided additional information for the same to the bank.
- A bank will update the KYC data if required.
- If the data is not already present with the Smart Contract, then the bank will create a new request to add the KYC of the customer.
- Banks will additionally provide upvotes/downvotes over the user data for KYC.

  Banks can also report the other banks for security and authenticity of the banks in the network.


Lists
Smart Contract over Blockchain also maintains a list of all Banks, Customer and Requests which are part of the system.
Blockchain
Creates Request
Bank stores the KYC docs at a secure storage and create a hash link for the docs to be added into the KYC request over the Smart Contract.
Customer
Bank
KYC DATA
Customer submits the KYC Data to the bank. This also includes personal information which will be used by bank to raise the KYC request.
Secure Storage

Bank Interface

You need to write the following functions in your smart contract. These are the functions that a bank can call.

- **Add Request -** This function is used to add the KYC request to the requests list.
    - **Parameters -** Customer name as a string and hash of the customer data as a string
- **Add Customer -** This function will add a customer to the customer list.
    - **Parameters** - Customer name as a string and customer data as a string
- **Remove Request** - This function will remove the request from the requests list.
    - **Parameters** - Customer name as a string

- **View Customer** - This function allows a bank to view the details of a customer.
    - **Parameter** - Customer name as a string
    - **Return** - All the variables of the customer structure
- **Upvote Customers -** This function allows a bank to cast an upvote for a customer. This vote from a bank means that it accepts the customer details as well as acknowledges the KYC process done by some bank for the customer.
    - **Parameter** - Customer name as a string
- **Downvote Customers -** This function allows a bank to cast a downvote for a customer. This vote from a bank means that it does not accept the customer details.
    - **Parameter** - Customer name as a string
- **Modify Customer** - This function allows a bank to modify a customer's data. This will remove the customer from the KYC request list and set the number of downvotes and upvotes to zero.
    - **Parameter** - Customer username and data as a string
- **Get Bank Complaints -** This function is used to fetch bank complaints from the smart contract.
    - **Parameter -** Bank address is passed as address to fetch bank ratings
    - **Return -** Integer number of complaintsReported against the bank
- **View Bank Details (Unique Identifier for the Bank)**
  - This function is used to fetch the bank details.
    - **Parameter -** Bank address is passed to the function
    - **Return -** The return type of this function will be of type Bank
- **Report Bank -** This function is used to report a complaint against any bank in the network. It will also modify the isAllowedToVote status of the bank according to the conditions mentioned in the problem statement.
    - **Parameter -** Bank address and bank name is passed to the function

Admin Interface

The following functions are specific to the admin:

- **Add Bank -** This function is used by the admin to add a bank to the KYC Contract. You need to verify whether the user trying to call this function is the admin or not.
    - **Parameters -** Bank name as string, bank address as address and bank registration number as string are passed to the function "Add Bank". Set the number of complaintsReported initially to zero and KYC permission as 'allowed/true'
- **Modify Bank isAllowedToVote -** This function can only be used by the admin to change the status of isAllowedToVote of any of the banks at any point in time.
    - **Parameters -** Bank address and the boolean value that is to be set to isAllowedToVote
- **Remove Bank -** This function is used by the admin to remove a bank from the KYC Contract. You need to verify whether the user trying to call this function is the admin or not.
    - **Parameter -** Bank address as address is passed to the function "Remove Bank"

Smart Contract Flow

- The bank collects the information for the KYC from the customer.
- The information given by the customer includes username and customer data, which is the hash of the link for the customer data. This username is unique for each customer.
- A bank creates the request for submission, which is stored in the smart contract.

- A bank then verifies the KYC data, which is then added to the customer list.
- Other banks can get customer information from the customer list.
- Other banks can also provide upvotes/downvotes on customer data to showcase the authenticity of the data. If the number of upvotes is greater than the number of downvotes, then the kycStatus of that customer is set to true. If a customer gets downvoted by one-third of the banks, then the kycStatus of the customer is changed to false even if the number of upvotes is more than that of downvotes. For such logic, there should be a minimum of 5 or 10 banks in the network.
- In short, there are two conditions to be checked: The number of upvotes and downvotes and whether the number of downvotes is greater than one-third of the total number of banks.
- Banks can also complain against other banks if they find the bank to be corrupt and if it is verifying false customers. Such corrupted banks will then be banned from upvoting/downvoting further. If more than one-third of the total banks in the network complain against a certain bank, then the bank will be banned (i.e., set isAllowedToVote to false of that corrupt bank.).

Points to remember while writing the smart contract:

- Make use of constructors and modifiers in the code. For example, use a constructor to set msg.sender as admin while deploying the smart contract, and use a modifier to restrict the use of admin functionalities.
- Implement some mathematical logic to assess whether a bank is faulty or not. For example, use the number of complaints against each bank. If the complaints exceed more than half (or one-third) of the total banks present in the network, set isAllowedToVote to false, or if the customer verified by a bank gets more than a threshold number of downvotes, then the bank gets banned from doing any further KYC. For such a system to work, there should be a certain minimum number of banks in the network.

- You need to add certain checks from your side to optimise the smart contract, such as check whether the customer exists before addKycrequest.
- Only valid banks should be allowed to add/modify customers.
- Try to avoid the use of arrays to store data. Use mappings instead. Also, avoid the use of any loops in the code.

The next part of this graded assignment is phase 3, whose details are mentioned in the next segment.

Deployment of Smart Contract on Private Network

Phase 3 details:

In this phase, we would go into building a private Ethereum chain and running the smart contract over it. This simulation will help you see the complete industry-specific project related to Ethereum. This will also help you learn how to set up a chain and deploy smart contracts over the same. Many industries require you to set up such solutions. For example, if you want to do a patient record-sharing smart contract between multiple hospitals, you can use this as a reference to build such solutions.

Flow for deployment

Initialise truffle for the creation of the folder structure to work with smart contracts

truffle is a solidity framework and IDE that is used to compile, deploy and test your smart contract. To start off, you first need to create a truffle project. The command given below will help you create the folder structure:

```
truffle init
```

Once the initialisation operation is complete, you will have a project structure with the following items:

- **contracts/ -** This is the directory for storing solidity contracts.
- **migrations/ -** This is the directory for scriptable deployment files, which, in turn, are Javascript files used for deploying contracts over the Ethereum network.
- **test/ -** This is the directory for storing the test files that are used for testing your application and contracts. These are based on Mocha or Chai frameworks.
- **truffle.js -** This is the truffle configuration file where you will provide your network details.

Create the KYC smart contract with truffle suite

You have already created the smart contract in phase 2. We will use the same smart contract in phase 3. You can create a file under the contracts folder, name it 'KYC.sol' and copy the contract to that file.

Note: .sol is the extension used for the solidity files, which are understandable by truffle suite.

You need to configure the truffle migrations to deploy the solidity code to a private blockchain. A sample migration file for the KYC Contract will look like this:

Filename: **2_kyc_migration.js**

```
var KYCContract = artifacts.require("KYCContract");

module.exports = function(deployer) {
  // deployment steps
  deployer.deploy(KYCContract);
};
```

If you notice here, the filename is prefixed with a number and suffixed by a description. The numbered prefix is required to record whether the migration ran successfully. The suffix is purely for human readability.

There are a few things to consider as mentioned below.

- **Artifacts.require -** This will tell truffle which contract we are working with.
- **Module.exports -** This is where we tell truffle to deploy the contract.
- **Deployer -** This is used to stage deployment tasks.

Update the truffle task runner with account and network details

After setting up the contract, you need to configure truffle for deployment on the private Ethereum chain. Your configuration file is called **truffle-config.js** and is located at the root of your project directory.

A sample configuration for the Blockchain will look like this:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1", // Match your private chain IP
      port: 8545, // Match your private chain port
      network_id: "2019" // Match you private chain network id
    }
  }
};
```

There are a few things to consider here as well. They are as follows:

- **Host -** This is the IP address for the node that is running the private blockchain. If you are deploying the smart contract from the same node, then you can use “127.0.0.1”, which diverts to the localhost. You can specify the host in your private chain by using the following command:

```
• geth --rpcaddr value
```

If you do not specify anything, then the default host is taken as “localhost”.

- **Port -** This is the port number that your private blockchain is utilising. You can specify the port in your private chain by using the following command:

```
• geth --rpcport value
```

If you do not specify anything, then the default port is taken as “8545”.

- **Network id -** This is the network id supplied as an integer that is used by your private blockchain. You can specify the network id in your private chain by using the following command:

```
• geth --networkid value
```

There are many network ids that are reserved for pre-existent chains. Make sure that you provide the network id, as the default network id is taken as “1”, which will connect your solution with the Ethereum mainnet.

Apart from this, we can also provide the Ethereum compiler version for solidity in the **“truffle.js”** file. To add the compiler version, you need to add the following details in module.exports under truffle.js:

```
compilers: {
   solc: {
     version: <string>

}

}
```

Compile the smart contract using truffle

Once you are ready with all the artefacts, you can deploy the contract over the private Ethereum chain. But, before that, you need to compile the KYC contract.

To compile the contract, go to the root of the directory where the project is located, and then, type the following command in the terminal:

```
truffle compile
```

When this command is executed for the first time, all the contracts under the “contracts” folder are compiled. Upon subsequent runs, truffle only compiles the contracts that are changed since the last compile.

Deploy on the private ethereum blockchain

The deployment of the KYC smart contract happens through the migrate command. To run the KYC migration, run the following:

```
truffle migrate
```

This will run all the migration files located within your project's migrations directory. If you have already run the migration command before and are running it again, then tcoruffle migrate will start execution from the last migration that was run, thereby running only newly created migration files. If no new migration files exist, truffle migrate will not perform any action at all.

Once the smart contract is deployed, we can use the geth command line to execute the functions in the smart contract and simulate the KYC process over Blockchain.