SDE Assignment 1

> ***Name: Utkarsh Thusoo***
>
> ***Roll no: M20AIE318***

Q1. Pick an example of a blockchain-based application and draw 4+1 views for at least 2 scenarios. Explain each of these views in 200 words each.

<u>Ans:</u>

1. Crypto Currency Transaction Handling

The idea is to implement a use case where we can both sell & buy or in this case send & receive crypto currency with any of the wallets there are. For sake of simplicity we will choose **WazirX** which is an Indian startup handling Buying, Selling & Trading Bitcoin, Ethereum, Ripple, Litecoin and more cryptocurrencies in India.

4 + 1 View Handling

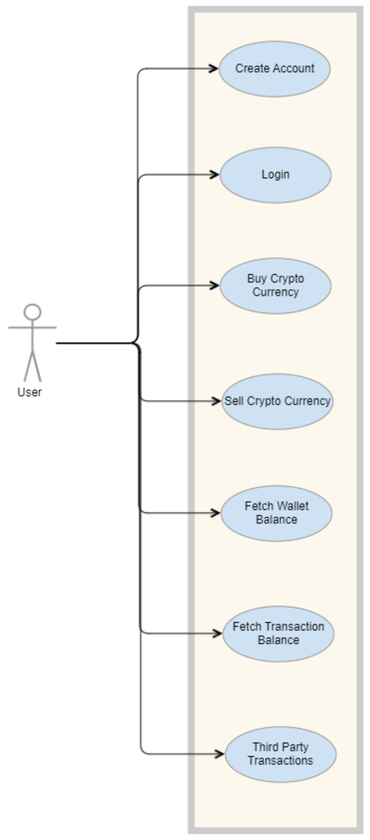
With this architectural design we try "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". For the given application we can define each of the view as following

Use Case

Below are some of the common use cases which can be implemented with **WaziX** when dealing with crypto currencies.

Sno	Use Case	As a user I should be allowed to...
1	Create a new Account	create register to the service by providing my details and create a new account.
2	Login to and existing account	access my account once I validate my identity
3	Buy Crypto currency using wallet	buy crypto currency and they should be stored in my wallet.
4	Sell Crypto currency using wallet	sell crypto currency already stored in my wallet
5	Make transactions to third party platforms to make other transactions	make transactions to third party platforms to make other transactions
6	View Balance in my wallet	view the balance available in my wallet
7	View previous transactions	view all transactions I have made using the wallet.

Use Case



Logical View

In logical view we are concerned with the functionality that the system provides to end-users. With **WaziX** we will try to represent the logical view with the help of class diagrams and state diagrams. We will try to include class diagrams covering all the use cases covered under use case scenario.

Class definition for given scenario

UserModel

User model defines all the data related to a specific user like his name, phone number, email, DOB, and a unique id. That unique ID can be used throughout the application to identify a user.

WalletModel

Wallet model defines all the data related to the wallet. This model includes :

- **userId** : Unique ID for each user
- **walletId**: Unique ID mapped to each *userId*.
- **balance**: Amount of crypto coins inside wallet for that *userId*.

TransactionModel

Transaction model defines all the data related to the wallet. This model includes:

- **fromUserId**: *userId* for the customer from where money will be debuted.
- **walletId**: *userId* for the customer in whose wallet money will be credited.
- **description**: Message description explaining transition details and status.
- **transactionId**: Unique ID to identify each transaction.
- **timestamp**: timestamp with both date and time explaining when the transaction was initiated.
- **status:** enum value representing what is the status of the current transaction. Status can be RUNNING/FAILED/NEW etc.

TransactionModelChain

This model represents the blockchain node which extends all the values from *TransactionModel* and defines other parameters like:

- **hash**: creates a unique hash value based on all the parameters present in the *TransactionModel* class.
- **previousHash**: stores the value of previous hash value of the transaction.

ICoreService

Interface which represents all the operations related to a user. The operations include:

- **createUser() :** abstract method which should handle creating a new user
- **getTransactions()** : abstract method which should handle fetching all transactions for a particular
- **buyCoins()** : abstract method which should handle buying coins
- **sellCoins()** : abstract method which should handle selling coins
- **getTransactionHistory()** : abstract method which should handle all transactions history
- **performKYC()** : abstract method which should handle KYC for that customer.

CoreImplentationService

Implementation class which represents all the operations from *ICoreService* interface.

ITransactionService

Interface which represents all the operations related to a transactions for that user. The operations include:

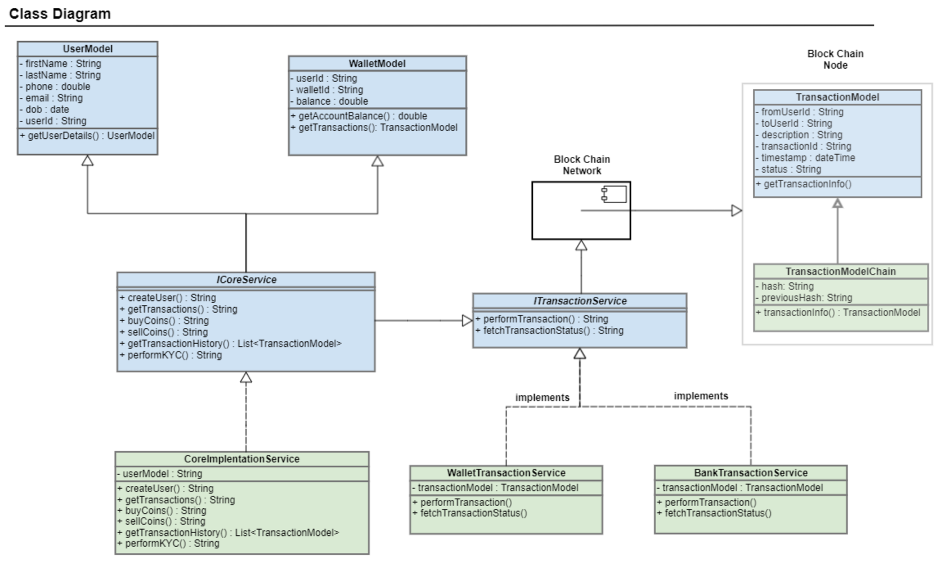
- **performTransaction() :** abstract method which should handle performing buying and selling transactions for user.
- **fetchTransactionStatus()** : abstract method which should handle fetching all transaction status for a particular transaction

WalletTransactionService

Implementation class which represents all the operations from *ITransactionService* interface from the wallet of the user.

BankTransactionService

Implementation class which represents all the operations from *ITransactionService* interface from the bank related transactions.



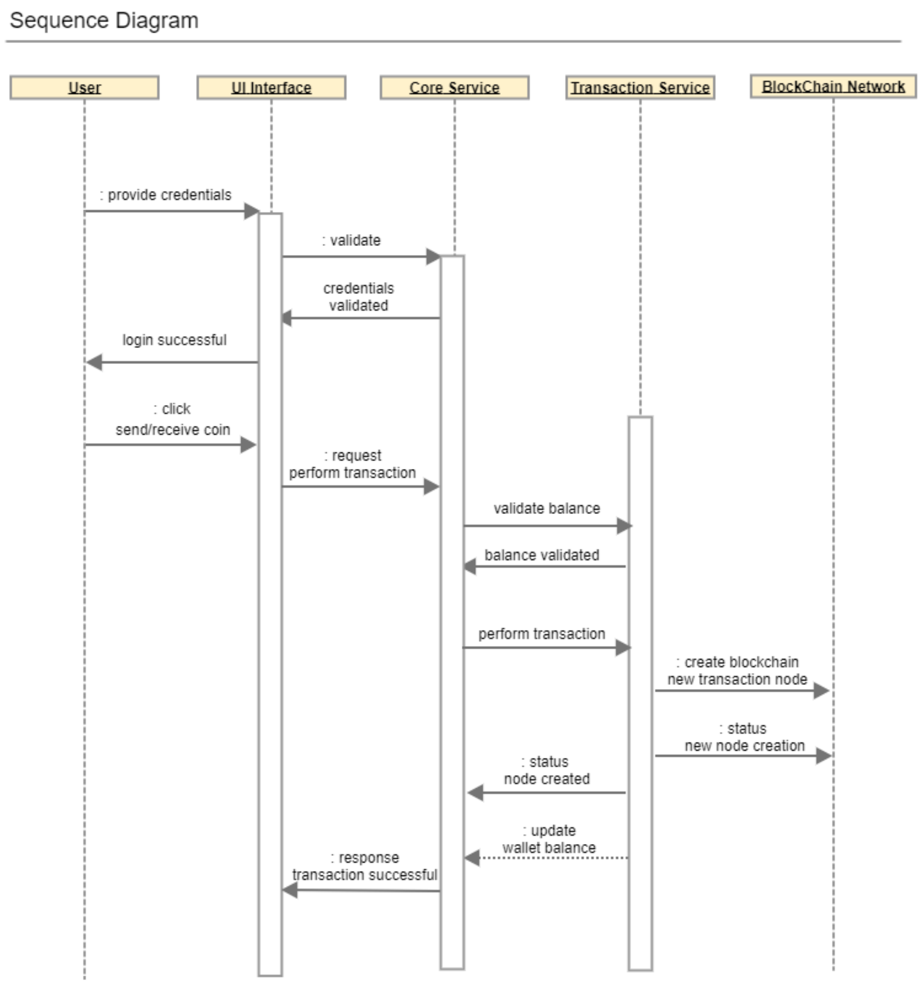
Process View

In process view we try to define dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system. In this case we can define a scenario where user wants to buy/sell crypto currency using his wallet. Below are the steps and a process diagram explaining this scenario:

Send bitcoins with wallet transactions

- *User* will login to the application using his credentials
- Credentials are validated and user is allowed to access his account.
- *User* then initiates a transaction from the UI to sell the bitcoins
- Request is initiated and is sent to backend *CoreService*
    - *CoreService* validates the requests
    - Checks the wallet balance
    - Creates a TransactionService call to perform the transaction

- *TransctionService* handles the bit coin transfer by
    - Creating a new node with unique *transactionId* in the blockchain network
    - validates if the node is created successfully
    - Updates the balance of the wallet if the transaction is successful
- Once transaction is successful send the feedback to *CoreService*
- Return response to UI that request is successful
- UI displays successful message to the user

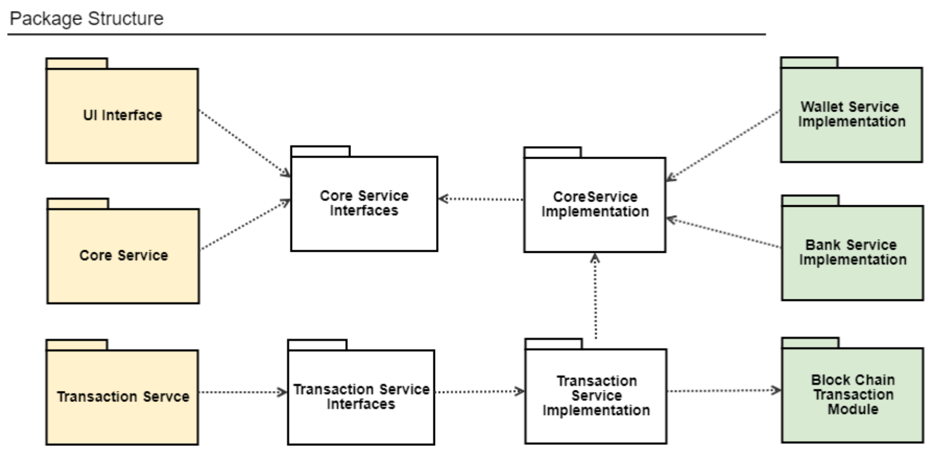


Development View

In development view we try to illustrate a system from a programmer's perspective and is concerned with software management. Here we will try to do the same with the help of package diagram.

Package definition for given scenario

Model	Module will hold all...
UI Interface	the packages related to UI
Core Service	core service classes like *CoreImplentationService* etc.
Transaction Service	services leading to transaction service.
Block Chain Transaction Service	interactions linked to block chain operations

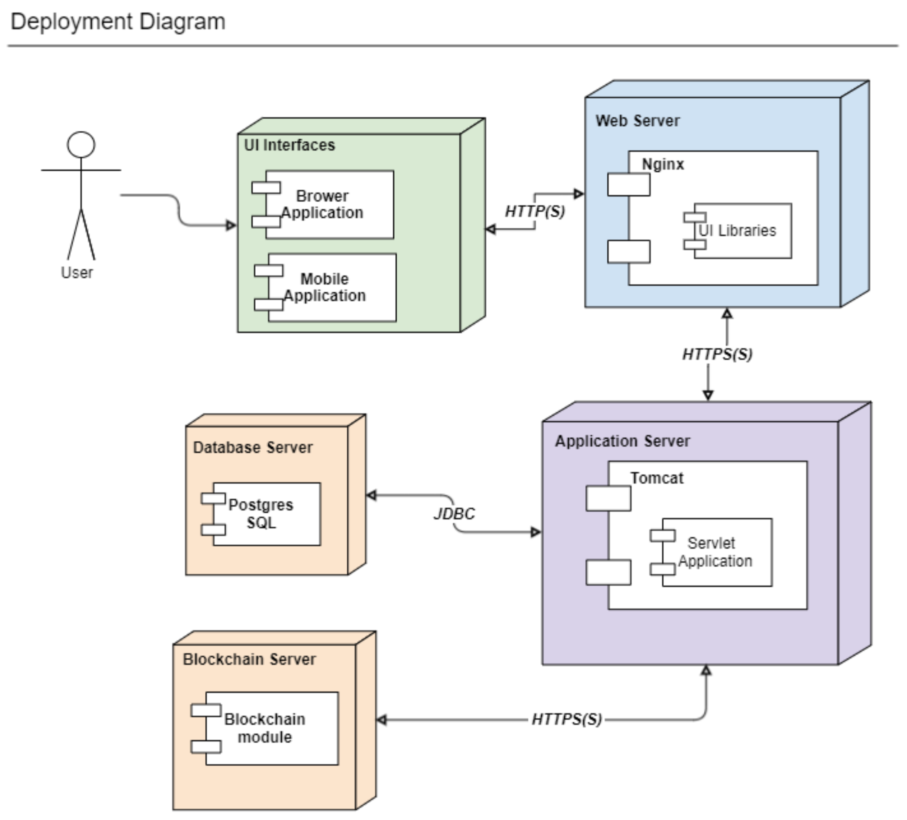

Package Structure
UI Interface
Core Service
Transaction Servce
Core Service Interfaces
Transaction Service Interfaces
CoreService Implementation
Transaction Service Implementation
Wallet Service Implementation
Bank Service Implementation
Block Chain Transaction Module

Physical View

In physical view we depict the system from a system engineer's point of view where we present topology of software components on the physical layer as well as the physical connections between these components. Deployment diagram for the same can be seen as under.

Deployment Diagram for given scenario

Model	**Module will hold all...**
UI Interface	UI interfaces available to customer • Website • Mobile Application
WebServer	Backend service for UI deployed on nginx
Application Server	All core services are deployed on ApplicationServer
Block Chain Server	block chain related node creation and handling
Database Server	all database related modules



2. KYC using blockchain application

The idea is to implement a use case where user can perform his KYC online based on a blockchain model. The customer can simply provide his credentials and relevant documents following which the details will be stored in a blockchain node. The main advantage here would be user won't be able to perform modifications without approval of multiple members from in the blockchain.

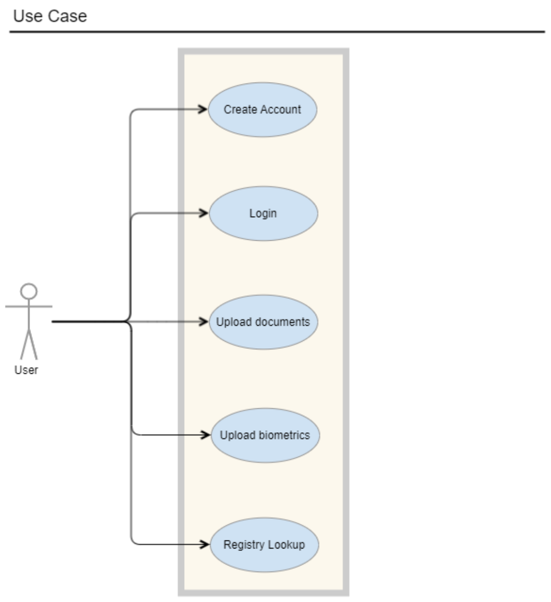
4 + 1 View Handling

With this architectural design we try "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". For the given application we can define each of the view as following

Use Case

Below are some of the common use cases which can be implemented online KYC approval with blockchain. One of such application is **KYC-Chain**

Sno	Use Case	As a user I should be allowed to...
1	Create a new Account	create register to the service by providing my details and create a new account.
2	Login to and existing account	access my account once I validate my identity
3	Upload documents to be validated	upload the documents which they want to be validated
4	Upload biometric details	upload all the biometric details for the user to be validated
5	Registry lookup	perform lookup for the customer documents in the registry



Logical View

Class definition for given scenario

UserModel

User model defines all the data related to a specific user like his name, phone number, email, DOB, and a unique id. That unique ID can be used throughout the application to identify a user.

KYCTypeModel

Defines the document, type which is uploaded to perform KYC. Will also include biometric data. This model includes :

- **userId** : Unique ID for each user
- **authName** : The name of the document/Biometric which was uploaded to perform KYC
- **authType**: The type of the document/Biometric which was uploaded to perform KYC
- **authData**: the document/biometric data converted to blob and stored
- **authStatus**: status of the document which identifies which status it is in. Can be *UPLOADED/PROCESSING/ACCEPTED/REJECTED*.

AuthInfoModel

Model defines the stage we are in where of the authentication process.

- **authenticationId** : unique id generated whenever an KYC is to be performed for a document.
- **timestamp** : time when authentication for initiated
- **userId** : Unique ID for each user
- **authName** : The name of the document/Biometric which was uploaded to perform KYC
- **authType**: The type of the document/Biometric which was uploaded to perform KYC
- **authData**: the document/biometric data converted to blob and stored
- **authStatus**: status of the document which identifies which status it is in. Can be *UPLOADED/PROCESSING/ACCEPTED/REJECTED*.

AuthInfoModelChain

This model represents the blockchain node which extends all the values from *AuthInfoModel* and defines other parameters like:

- **hash**: creates a unique hash value based on all the parameters present in the *AuthInfoModel* class.
- **previousHash**: stores the value of previous hash value of the transaction.

ICoreService

Interface which represents all the operations related to a user. The operations include:

- **createUser() :** abstract method which should handle creating a new user
- **getTransactions()** : abstract method which should handle fetching all transactions for a particular
- **buyCoins()** : abstract method which should handle buying coins

- **sellCoins()** : abstract method which should handle selling coins
- **getTransactionHistory()** : abstract method which should handle all transactions history
- **performKYC()** : abstract method which should handle KYC for that customer.
- **createUser() :** abstract method which should handle creating a new user: String
- **isPersonAvailable() :** abstract method which should check a person is already available for the KYC document provided
- **uploadDocs()** : abstract method which should perform document upload operation
- **getKYCStatus()** : abstract method which should get KYC details symptoms
- **performKYC()** : abstract method which should initiate KYC for the document uploaded
- **getAuthDoc():** abstract method which should return an authentication model asked for
- **getAuthDocs():** abstract method which should return all the documents for the user
- **updateDoc()** : abstract method which should update a document detail for the user.

CoreImplentationService

Implementation class which represents all the operations from *ICoreService* interface.

IAuthenticationService

Interface which represents all the operations related to a authentication for that user. The operations include:

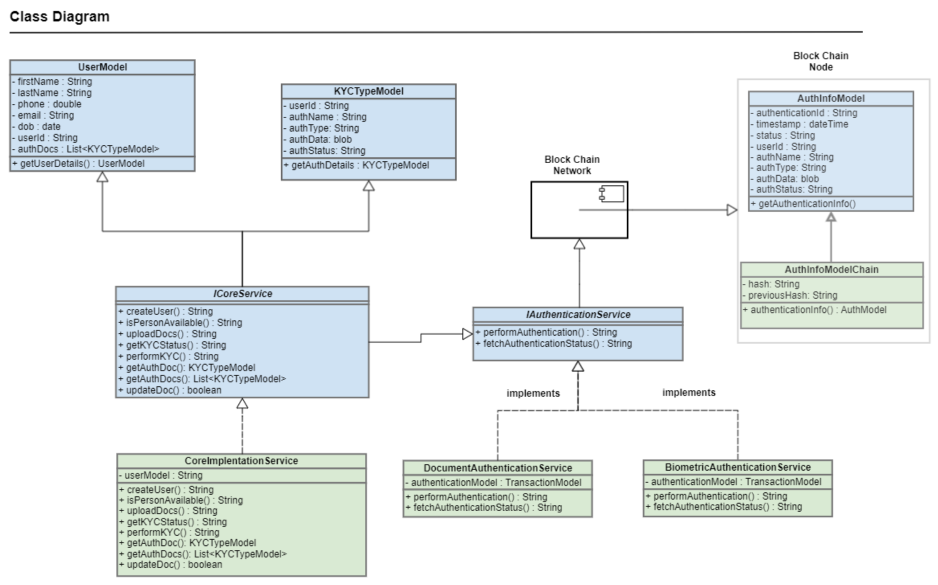
- **performAuthentication() :** abstract method which should handle authentication process for the KYC document uploaded.
- **fetchAuthenticationStatus()** : abstract method which should handle fetching all authentication status for a particular document

DocumentAuthenticationService

Implementation class which represents all the operations from *IAuthenticationService* interface based on the document uploaded by the user.

BiometricAuthenticationService

Implementation class which represents all the operations from *IAuthenticationService* interface based on the biometric data uploaded by the user.


Class Diagram
UserModel
- firstName : String
- lastName : String
- phone : double
- email : String
- dob : date
- userId : String
- authDocs : List<KYCTypeModel>
+ getUserDetails() : UserModel
KYCTypeModel
- userId : String
- authName : String
- authType: String
- authData: blob
- authStatus: String
+ getAuthDetails : KYCTypeModel
Block Chain Network
Block Chain Node
AuthInfoModel
- authenticationId : String
- timestamp : dateTime
- status : String
- userId : String
- authName : String
- authType: String
- authData: blob
- authStatus: String
+ getAuthenticationInfo()
AuthInfoModelChain
- hash: String
- previousHash: String
+ authenticationInfo() : AuthModel
ICoreService
+ createUser() : String
+ isPersonAvailable() : String
+ uploadDocs() : String
+ getKYCStatus() : String
+ performKYC() : String
+ getAuthDoc(): KYCTypeModel
+ getAuthDocs(): List<KYCTypeModel>
+ updateDoc() : boolean
IAuthenticationService
+ performAuthentication() : String
+ fetchAuthenticationStatus() : String
implements
implements
CoreImplentationService
- userModel : String
+ createUser() : String
+ isPersonAvailable() : String
+ uploadDocs() : String
+ getKYCStatus() : String
+ performKYC() : String
+ getAuthDoc(): KYCTypeModel
+ getAuthDocs(): List<KYCTypeModel>
+ updateDoc() : boolean
DocumentAuthenticationService
- authenticationModel : TransactionModel
+ performAuthentication() : String
+ fetchAuthenticationStatus() : String
BiometricAuthenticationService
- authenticationModel : TransactionModel
+ performAuthentication() : String
+ fetchAuthenticationStatus() : String

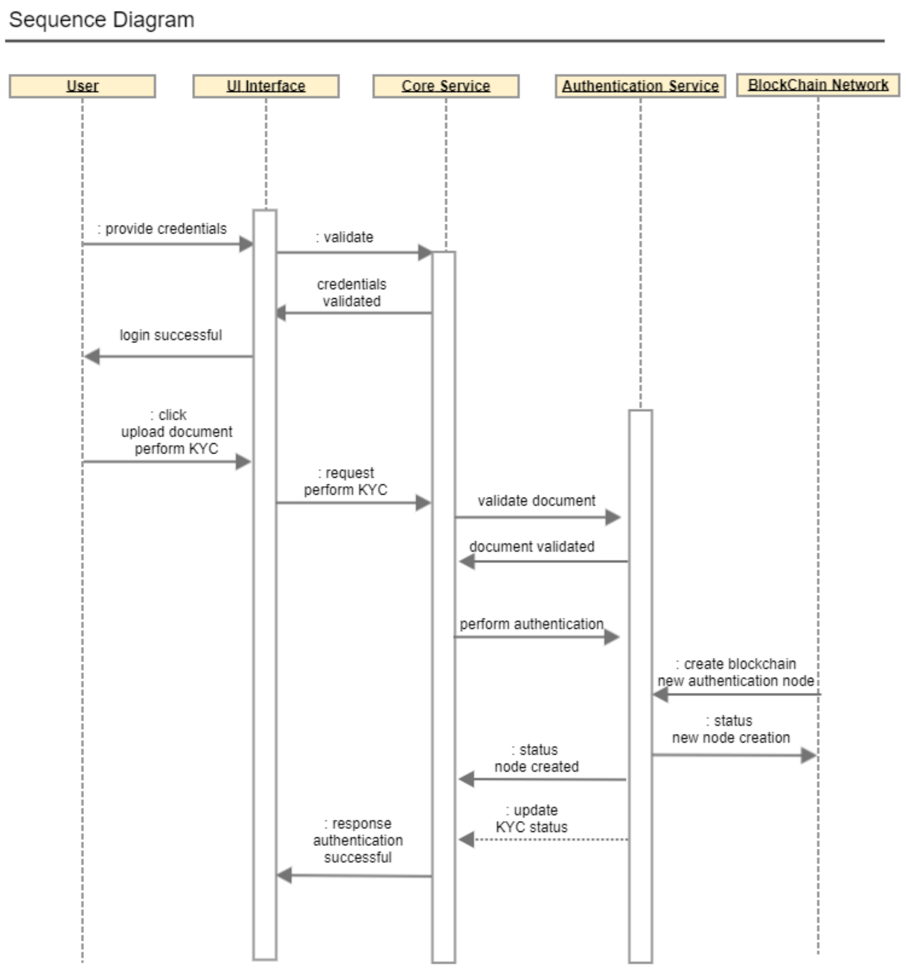
Process View

In process view we try to define dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system. In this case we can define a scenario where user perform KYC for the document uploaded. Below are the steps and a process diagram explaining this scenario:

Perform KYC for the uploaded document

- *User* will login to the application using his credentials
- Credentials are validated and user is allowed to access his account.
- *User* uploads document to perform KYC and initiates authentication process
- Request is initiated and is sent to backend *CoreService*
    - *CoreService* validates the requests
    - Check if document is already uploaded
    - Creates a AuthenticationService call to perform the authentication
- *AuthenticationService* handles the authentication process
    - Creating a new node with unique *authenticationId* in the blockchain network
    - validates if the node is created successfully

    - Updates the balance of the wallet if the authentication is successful
- Once authentication is successful send the feedback to *CoreService*
- Return response to UI that request is successful
- UI displays successful message to the user

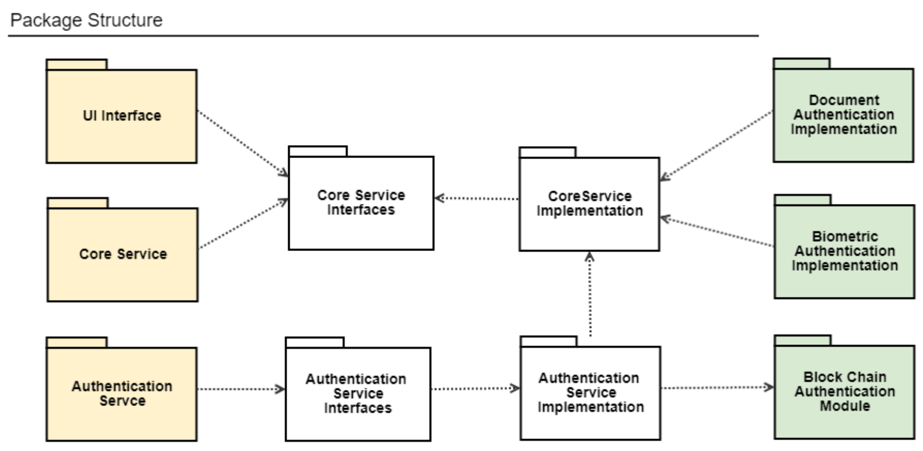


Development View

In development view we try to illustrate a system from a programmer's perspective and is concerned with software management. Here we will try to do the same with the help of package diagram.

Package definition for given scenario

Model	Module will hold all...
UI Interface	the packages related to UI
Core Service	core service classes like *CoreImplentationService* etc.
Authentication Service	services leading to authentication service.
Block Chain Authentication Service	interactions linked to block chain operations

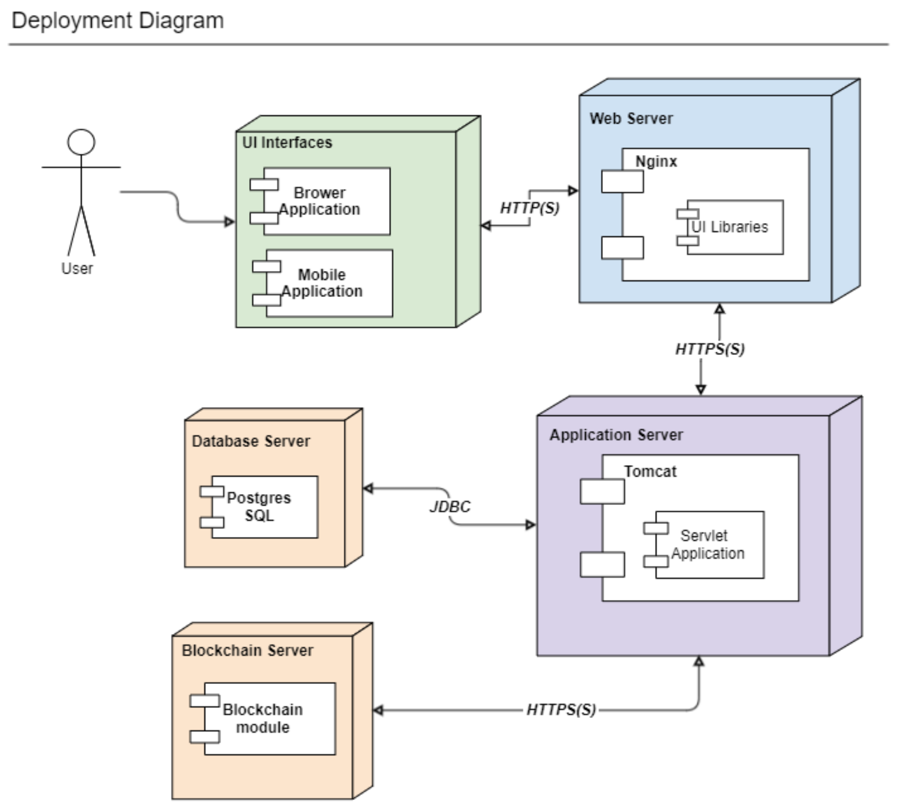

Package Structure
UI Interface
Core Service
Authentication Servce
Core Service Interfaces
Authentication Service Interfaces
CoreService Implementation
Authentication Service Implementation
Document Authentication Implementation
Biometric Authentication Implementation
Block Chain Authentication Module

Physical View

In physical view we depict the system from a system engineer's point of view where we present topology of software components on the physical layer as well as the physical connections between these components. Deployment diagram for the same can be seen as under.

Deployment Diagram for given scenario

Model	**Module will hold all...**
UI Interface	UI interfaces available to customer • Website • Mobile Application
WebServer	Backend service for UI deployed on nginx
Application Server	All core services are deployed on ApplicationServer
Block Chain Server	block chain related node creation and handling
Database Server	all database related modules



Q2. Identify two design patterns used in the app and explain each of the design patterns with the following subsections

a) Problem Statement

b) Context

c) Forces/Constraints and

d) Solutions - Static & Dynamic schematics

e) Consequences.

Ans:

1. Singleton Pattern

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

a) Problem Statement

Creating a new uniqueId whenever asked. the Unique ID can be authentication ID, user ID etc.

b) Context

Whenever a new user is created we need to identify the user uniquely in the application. For generation we need to create a thread-safe method which when called returns a unique ID every time.

c) Constraints

Generation of ID for multi threaded call might take longer than usual.

d) Solution

```
private static synchronized long get64MostSignificantBitsForVersion1() {
    LocalDateTime start = LocalDateTime.of(1582, 10, 15, 0, 0, 0);
    Duration duration = Duration.between(start, LocalDateTime.now());
    long seconds = duration.getSeconds();
    long nanos = duration.getNano();
    long timeForUuidIn100Nanos = seconds * 10000000 + nanos * 100;
    long least12SignificatBitOfTime = (timeForUuidIn100Nanos & 0x000000000000FFFFL) >> 4;
    long version = 1 << 12;
    return
      (timeForUuidIn100Nanos & 0xFFFFFFFFFFFF0000L) + version + least12SignificatBitOfTime;
}
```

e) Consequences

We need to implement synchronized which might iterate delay.

2. Factory Pattern

is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

a) Problem Statement

Initialize the transaction object based on the type of transaction to be performed. If the customer wants to perform wallet based transaction depending on the type of the transaction i.e. wallet transactions need to performed by a different class and bank transactions need to be performed by a spearate class.

b) Context

We can perform transactions either from a wallet or using a bank when buying or selling crypto currency. For the same we can use abstract factory pattern where interface can be *ITransactionService* with corresponding implementation classes implemented under *WalletTransactionService* & *BankTransactionService*.

c) Constraints

We need to be careful while initializing the object and it is preferred if this is a lazy initialization.

d) Solution

```
import java.io.*;
interface ITransactionService{
        String performTransaction();
        String fetchTransactionStatus();
}


class WalletTransactionService implements ITransactionService{
         private final String tType;
         public WalletTransactionService(){
                tType="WALLET";
        }
        public String getTransactionType() {
                  return tType;
        }
}
class BankTransactionService implements ITransactionService{
       private final String tType;
       BankTransactionService(){
               tType="BANK";
        }
        public String getTransactionType() {
                  return tType;

  }
}


class ITransactionServiceFactory{
   public ITransactionService getService(String tType){
      if(tType == null){
         return null;
      }
      if(tType.equalsIgnoreCase("WALLET")){
         return new BankTransactionService();
      } else if(tType.equalsIgnoreCase("BANK")){
         return new WalletTransactionService();
      }
      return null;
   }

}
```

e) Consequences

This will help us in creating the object and the approach is scalable because a new Transaction type can be easily added.