The approach for generating chaincode using ChatGPT for end-to-end chaincode involves several key steps and considerations:

Understanding Specific Requirements: The process begins with a clear understanding of the requirements for the chaincode. This involves identifying the specific business logic and network details that the chaincode needs to address. For instance, the supply chain management chaincode is designed to track packages from origin to delivery, including payment release mechanisms​​.

Defining Data Structures and Chaincode Functions: The chaincode is structured around key data structures and functions relevant to the application. For example, in the supply chain management scenario, structures like Asset, Handler, and Payment are defined, along with functions to create and update assets, acknowledge receipt, and release payments​​.

Tailoring to Specific Industries: The chaincode is customized for specific industry needs, such as pharmaceuticals. This involves adding features like batch numbers, expiry dates, storage conditions, recall handling, and counterfeit protection​​.

Event Handling and Additional Scenarios: Implementing event handling for various conditions like package loss, damage, or holding issues, and addressing additional scenarios specific to the industry, such as storage condition validations and handling counterfeit medications​​.

Incorporating Token and NFT Functionality: Extending the chaincode to include functionalities for managing tokens and non-fungible tokens (NFTs). This includes creating and managing token balances, minting and transferring tokens or NFTs, and querying balances or NFT details, similar to functionalities in blockchain systems like Ethereum​​.

Input Assumptions and Validations: Making assumptions about the format of key inputs and implementing validations to ensure that the data provided to the chaincode meets expected standards. This step is crucial for maintaining data integrity and smooth operation of the chaincode​​.

Adapting to Real-World Applications: Recognizing that the provided chaincode examples are basic representations and need further details and robustness for real-world applications. This includes considering more granular details, business-specific logic, intricate condition checks, and external system integrations​​.

We evaluated the generated chaincodes by using the following test cases and scenarios:

* We generated eight different chaincodes named DEX AMM, DEX Orderbook, First Price Bid, Mint NFT Supplychain, NFT Collectibles, NFT Event Ticket, NFT Real Estate, and NFT Second Price Bid by providing ChatGPT with the necessary information and instructions for each chaincode.
* We copied the generated chaincodes into a “Hyperledger Fabric” project generated by “Hyperledger Fablo”. Fablo had “Fablo REST” and “Blockchain Explorer” set up for the project.
* We used the APIs for each chaincode generated by Fablo REST to test the functionality and performance of the chaincodes. We checked the input and output parameters, the response time, and the error handling of the APIs.
* We used Blockchain Explorer to test the generated transactions and blocks for each successful API call. We verified the transaction ID, the block number, the payload, and the endorsement policy of the transactions and blocks.

Below are the chaincodes:

1. NFT Collectibles

I updated the previous chaincode with a use case:

\_\_\_\_\_\_\_\_

package chaincode

import (

"encoding/json"

"fmt"

"strconv"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

// SmartContract defines the structure of the chaincode

type SmartContract struct {

contractapi.Contract

}

// NFT represents an art collectible with details

type NFT struct {

ID string `json:"id"`

Owner string `json:"owner"`

Price int `json:"price"` // Price is in FabBits (1 FabCoin = 1000 FabBits)

}

// InitLedger initializes the ledger with some default values (if needed)

func (t \*SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {

nfts := []NFT{

{ID: "NFT1", Owner: "Alice", Price: 10000},

{ID: "NFT2", Owner: "Bob", Price: 15000},

}

for \_, nft := range nfts {

nftBytes, err := json.Marshal(nft)

if err != nil {

return err

}

err = ctx.GetStub().PutState(nft.ID, nftBytes)

if err != nil {

return err

}

}

return nil

}

// Mint creates a new NFT in the ledger

func (t \*SmartContract) Mint(ctx contractapi.TransactionContextInterface, id string, priceInFabCoin float64) error {

if priceInFabCoin <= 0 {

return fmt.Errorf("price must be a positive value")

}

priceInFabBits := int(priceInFabCoin \* 1000) // Convert FabCoin to FabBits

ownerId, err := ctx.GetClientIdentity().GetID()

if err != nil {

return fmt.Errorf("failed to get client identity: %s", err)

}

nft := NFT{

ID: id,

Owner: ownerId,

Price: priceInFabBits,

}

nftBytes, err := json.Marshal(nft)

if err != nil {

return fmt.Errorf("failed to marshal NFT: %s", err)

}

return ctx.GetStub().PutState(id, nftBytes)

}

// Transfer changes the ownership of an NFT

func (t \*SmartContract) Transfer(ctx contractapi.TransactionContextInterface, id string, newOwner string) error {

nftBytes, err := ctx.GetStub().GetState(id)

if err != nil {

return fmt.Errorf("failed to get NFT: %s", err)

}

if nftBytes == nil {

return fmt.Errorf("NFT not found")

}

nft := NFT{}

err = json.Unmarshal(nftBytes, &nft)

if err != nil {

return fmt.Errorf("failed to unmarshal NFT: %s", err)

}

nft.Owner = newOwner

updatedNftBytes, err := json.Marshal(nft)

if err != nil {

return fmt.Errorf("failed to marshal updated NFT: %s", err)

}

return ctx.GetStub().PutState(id, updatedNftBytes)

}

// PurchaseNFT allows a user to buy an NFT from another user

func (t \*SmartContract) PurchaseNFT(ctx contractapi.TransactionContextInterface, nftID string, paymentInFabCoin float64) error {

paymentInFabBits := int(paymentInFabCoin \* 1000)

buyerID, err := ctx.GetClientIdentity().GetID()

if err != nil {

return fmt.Errorf("failed to get buyer identity: %s", err)

}

nftBytes, err := ctx.GetStub().GetState(nftID)

if err != nil {

return fmt.Errorf("failed to get NFT: %s", err)

}

if nftBytes == nil {

return fmt.Errorf("NFT not found")

}

nft := NFT{}

err = json.Unmarshal(nftBytes, &nft)

if err != nil {

return fmt.Errorf("failed to unmarshal NFT: %s", err)

}

if buyerID == nft.Owner {

return fmt.Errorf("buyer cannot be the current owner")

}

if paymentInFabBits < nft.Price {

return fmt.Errorf("insufficient payment, NFT price is %d FabBits", nft.Price)

}

// Logic to deduct FabBits from buyer's account and add to seller's account

// [Assuming the existence of account management logic in the chaincode]

nft.Owner = buyerID

updatedNftBytes, err := json.Marshal(nft)

if err != nil {

return fmt.Errorf("failed to marshal updated NFT: %s", err)

}

return ctx.GetStub().PutState(nftID, updatedNftBytes)

}

// QueryNFT retrieves details of an NFT

func (t \*SmartContract) QueryNFT(ctx contractapi.TransactionContextInterface, nftID string) (\*NFT, error) {

nftBytes, err := ctx.GetStub().GetState(nftID)

if err != nil {

return nil, fmt.Errorf("failed to get NFT: %s", err)

}

if nftBytes == nil {

return nil, fmt.Errorf("NFT not found")

}

nft := new(NFT)

err = json.Unmarshal(nftBytes, nft)

if err != nil {

return nil, fmt.Errorf("failed to unmarshal NFT: %s", err)

}

return nft, nil

}

// SetAccountBalance sets the balance of a given account (for testing)

func (t \*SmartContract) SetAccountBalance(ctx contractapi.TransactionContextInterface, accountID string, balanceInFabCoin float64) error {

balanceInFabBits := int(balanceInFabCoin \* 1000)

return ctx.GetStub().PutState(accountID, []byte(strconv.Itoa(balanceInFabBits)))

}

// main function can be uncommented when deploying the chaincode

// func main() {

// chaincode, err := contractapi.NewChaincode(&SmartContract{})

// if err != nil {

// fmt.Printf("Error creating NFT chaincode: %s", err.Error())

// return

// }

// if err := chaincode.Start(); err != nil {

// fmt.Printf("Error starting NFT chaincode: %s", err.Error())

// }

// }

### **Testing the Chaincode:**

* **Deploy the Chaincode**: Use the main function to deploy this chaincode to your Hyperledger Fabric network.
* **Initialize the Ledger**: The InitLedger function will create two NFTs, one owned by "Alice" and the other by "Bob".
* **Create and Manage Users**: Utilize SetAccountBalance to simulate user account balances for transactions.
* **Mint, Transfer, and Purchase NFTs**: Test these functionalities using the respective functions (Mint, Transfer, PurchaseNFT) with various scenarios.
* **Query NFTs**: Use the QueryNFT function to inspect NFT details post-transaction.

1. NFT real estate

package main

import (

"encoding/json"

"fmt"

"strconv"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

type SmartContract struct {

contractapi.Contract

}

// PropertyNFT and PropertyShare struct definitions remain unchanged

// Initialize the Smart Contract with sample data

func (s \*SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {

properties := []PropertyNFT{

{PropertyID: "Prop001", Description: "Luxury Apartment in New York", TotalShares: 1000},

{PropertyID: "Prop002", Description: "Beach House in California", TotalShares: 500},

}

for \_, property := range properties {

propertyAsBytes, \_ := json.Marshal(property)

err := ctx.GetStub().PutState(property.PropertyID, propertyAsBytes)

if err != nil {

return fmt.Errorf("failed to put to world state. %v", err)

}

}

return nil

}

// Mint a new PropertyNFT

func (s \*SmartContract) MintPropertyNFT(ctx contractapi.TransactionContextInterface, args []string) error {

if len(args) != 3 {

return fmt.Errorf("incorrect number of arguments. expecting 3")

}

propertyID := args[0]

description := args[1]

totalShares, err := strconv.Atoi(args[2])

if err != nil {

return fmt.Errorf("invalid total shares value")

}

property := PropertyNFT{

PropertyID: propertyID,

Description: description,

TotalShares: totalShares,

}

propertyAsBytes, \_ := json.Marshal(property)

return ctx.GetStub().PutState(propertyID, propertyAsBytes)

}

// Buy shares in a PropertyNFT

func (s \*SmartContract) BuyShares(ctx contractapi.TransactionContextInterface, shareID, propertyID, buyer string, numShares, priceInCrypto int) error {

propertyAsBytes, err := ctx.GetStub().GetState(propertyID)

if err != nil {

return fmt.Errorf("failed to get property: %s", err.Error())

}

if propertyAsBytes == nil {

return fmt.Errorf("property not found")

}

var property PropertyNFT

err = json.Unmarshal(propertyAsBytes, &property)

if err != nil {

return fmt.Errorf("failed to unmarshal property: %s", err.Error())

}

if numShares < 1 || numShares > property.TotalShares {

return fmt.Errorf("invalid number of shares requested")

}

// Create and store the property share

share := PropertyShare{

ShareID: shareID,

PropertyID: propertyID,

Owner: buyer,

NumShares: numShares,

}

shareAsBytes, \_ := json.Marshal(share)

return ctx.GetStub().PutState(shareID, shareAsBytes)

}

// Transfer shares from one owner to another

func (s \*SmartContract) TransferShares(ctx contractapi.TransactionContextInterface, shareID, currentOwner, newOwner string) error {

shareAsBytes, err := ctx.GetStub().GetState(shareID)

if err != nil {

return fmt.Errorf("failed to get property share: %s", err.Error())

}

if shareAsBytes == nil {

return fmt.Errorf("property share not found")

}

var share PropertyShare

err = json.Unmarshal(shareAsBytes, &share)

if err != nil {

return fmt.Errorf("failed to unmarshal property share: %s", err.Error())

}

if share.Owner != currentOwner {

return fmt.Errorf("transfer of shares can only be initiated by the current owner")

}

share.Owner = newOwner

updatedShareAsBytes, \_ := json.Marshal(share)

return ctx.GetStub().PutState(shareID, updatedShareAsBytes)

}

// Main function

func main() {

chaincode, err := contractapi.NewChaincode(new(SmartContract))

if err != nil {

fmt.Printf("Error creating new Smart Contract: %s", err)

return

}

if err := chaincode.Start(); err != nil {

fmt.Printf("Error starting Smart Contract: %s", err)

}

}

### **Testing the Chaincode:**

* **Populate Accounts**: Before testing, make sure both Alice and Bob have their account balances set up in the ledger.
* **Perform Transactions**: Test buying shares, ensuring the ledger accurately reflects the changes in ownership and account balances.

1. DEX AMM

​​

Did not return expected result

package chaincode

import (

"encoding/json"

"errors"

"math"

"time"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

type SmartContract struct {

contractapi.Contract

}

type LiquidityPool struct {

TokenA float64 `json:"tokenA"`

TokenB float64 `json:"tokenB"`

LastTransaction int64 `json:"lastTx"`

LiquidityLocked bool `json:"liquidityLocked"`

LockedUntil int64 `json:"lockedUntil"`

TotalLPTokens float64 `json:"totalLPTokens"`

}

const (

MAX\_SLIPPAGE = 0.01

TRANSACTION\_FEE = 0.001

MINIMUM\_LIQUIDITY = 10

RATE\_LIMIT\_SECONDS = 10

MIN\_LP\_TOKENS = 1

)

func (s \*SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {

pool := LiquidityPool{

TokenA: 10000,

TokenB: 10000,

LastTransaction: time.Now().Unix(),

LiquidityLocked: false,

LockedUntil: 0,

TotalLPTokens: 0,

}

poolBytes, \_ := json.Marshal(pool)

return ctx.GetStub().PutState("pool", poolBytes)

}

func (s \*SmartContract) AddLiquidity(ctx contractapi.TransactionContextInterface, amountA float64, amountB float64, provider string) (float64, error) {

poolBytes, err := ctx.GetStub().GetState("pool")

if err != nil || poolBytes == nil {

return 0, errors.New("pool not found")

}

var pool LiquidityPool

err = json.Unmarshal(poolBytes, &pool)

if err != nil {

return 0, errors.New("error unmarshalling pool data")

}

// Add tokens to the pool

pool.TokenA += amountA

pool.TokenB += amountB

// Calculate LP tokens to issue - proportional to the increase in liquidity

var issuedLPTokens float64

if pool.TotalLPTokens == 0 {

issuedLPTokens = math.Sqrt(amountA \* amountB) // Initial mint

pool.TotalLPTokens = issuedLPTokens

} else {

issuedLPTokens = math.Min(amountA/pool.TokenA, amountB/pool.TokenB) \* pool.TotalLPTokens

pool.TotalLPTokens += issuedLPTokens

}

// Update pool data

poolBytes, \_ = json.Marshal(pool)

err = ctx.GetStub().PutState("pool", poolBytes)

if err != nil {

return 0, errors.New("failed to update pool data")

}

return issuedLPTokens, nil

}

func (s \*SmartContract) Swap(ctx contractapi.TransactionContextInterface, amountA float64, maxSlippage float64, swapper string) (float64, error) {

poolBytes, err := ctx.GetStub().GetState("pool")

if err != nil || poolBytes == nil {

return 0, errors.New("pool not found")

}

var pool LiquidityPool

err = json.Unmarshal(poolBytes, &pool)

if err != nil {

return 0, errors.New("error unmarshalling pool data")

}

// Calculate amount of TokenB to be swapped

k := pool.TokenA \* pool.TokenB // Constant product

newTokenA := pool.TokenA + amountA

newTokenB := k / newTokenA

amountB := pool.TokenB - newTokenB

// Apply slippage check

actualSlippage := (amountB / amountA) - 1

if math.Abs(actualSlippage) > maxSlippage {

return 0, errors.New("slippage too high")

}

// Apply transaction fee

fee := amountB \* TRANSACTION\_FEE

amountB -= fee

// Update pool data

pool.TokenA = newTokenA

pool.TokenB = newTokenB

pool.LastTransaction = time.Now().Unix()

poolBytes, \_ = json.Marshal(pool)

err = ctx.GetStub().PutState("pool", poolBytes)

if err != nil {

return 0, errors.New("failed to update pool data")

}

return amountB, nil

}

func (s \*SmartContract) RemoveLiquidity(ctx contractapi.TransactionContextInterface, lpTokens float64, provider string) (float64, float64, error) {

if lpTokens < MIN\_LP\_TOKENS {

return 0, 0, errors.New("minimum LP tokens not met for withdrawal")

}

poolBytes, err := ctx.GetStub().GetState("pool")

if err != nil || poolBytes == nil {

return 0, 0, errors.New("pool not found")

}

var pool LiquidityPool

err = json.Unmarshal(poolBytes, &pool)

if err != nil {

return 0, 0, errors.New("error unmarshalling pool data")

}

// Calculate the amount of TokenA and TokenB to withdraw

share := lpTokens / pool.TotalLPTokens

amountA := share \* pool.TokenA

amountB := share \* pool.TokenB

// Update pool data

pool.TokenA -= amountA

pool.TokenB -= amountB

pool.TotalLPTokens -= lpTokens

poolBytes, \_ = json.Marshal(pool)

err = ctx.GetStub().PutState("pool", poolBytes)

if err != nil {

return 0, 0, errors.New("failed to update pool data")

}

return amountA, amountB, nil

}

// Additional helper functions or modifications can be added here

// func main() {

// err := shim.Start(new(SmartContract))

// if err != nil {

// fmt.Printf("Error creating new Smart Contract: %s", err)

// }

// }

### **Testing Steps:**

* **Initialization**: Start by initializing the ledger using InitLedger.
* **Adding Liquidity**:
  + Invoke AddLiquidity for John (AddLiquidity(ctx, 5000, 5000, "John")) and Dan (AddLiquidity(ctx, 3000, 3000, "Dan")).
* **Token Swapping**:
  + Alice swaps tokens using Swap(ctx, 100, 0.01, "Alice").
  + Bob swaps tokens using Swap(ctx, 150, 0.01, "Bob").
* **Removing Liquidity**:
  + John removes liquidity using RemoveLiquidity(ctx, 2000, "John").
  + Dan removes liquidity using RemoveLiquidity(ctx, 1000, "Dan").

1. DEX\_ orderbook

Did not return expected result

package main

import (

"encoding/json"

"fmt"

"time"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

// Asset, Order, User, OrderMatchedEvent struct definitions remain unchanged

func (s \*SmartContract) PlaceOrder(ctx contractapi.TransactionContextInterface, orderID, assetID, userID, orderType string, price, quantity float64, expiry int64) error {

// Validate order type

if orderType != "buy" && orderType != "sell" {

return ErrInvalidOrderType

}

// Check for valid quantity and price

if quantity <= 0 || price <= 0 {

return fmt.Errorf("invalid quantity or price")

}

// Create and store the order

order := Order{

OrderID: orderID,

AssetID: assetID,

UserID: userID,

OrderType: orderType,

Price: price,

Quantity: quantity,

Status: "open",

Timestamp: time.Now().Unix(),

Expiry: expiry,

}

orderJSON, err := json.Marshal(order)

if err != nil {

return fmt.Errorf("error marshalling order: %s", err)

}

err = ctx.GetStub().PutState(orderID, orderJSON)

if err != nil {

return fmt.Errorf("error storing order: %s", err)

}

// Attempt to match the order with existing orders

return s.matchOrder(ctx, &order)

}

func (s \*SmartContract) CancelOrder(ctx contractapi.TransactionContextInterface, orderID string) error {

// Retrieve the order

orderJSON, err := ctx.GetStub().GetState(orderID)

if err != nil {

return fmt.Errorf("error retrieving order: %s", err)

}

if orderJSON == nil {

return fmt.Errorf("order not found: %s", orderID)

}

var order Order

err = json.Unmarshal(orderJSON, &order)

if err != nil {

return fmt.Errorf("error unmarshalling order: %s", err)

}

// Check if the order can be cancelled

if order.Status != "open" {

return fmt.Errorf("only open orders can be cancelled")

}

if time.Now().Unix() > order.Expiry {

return ErrOrderExpired

}

// Cancel the order

order.Status = "cancelled"

updatedOrderJSON, \_ := json.Marshal(order)

return ctx.GetStub().PutState(orderID, updatedOrderJSON)

}

func (s \*SmartContract) matchOrder(ctx contractapi.TransactionContextInterface, order \*Order) error {

// Placeholder logic for matching orders

// In a real implementation, this would involve:

// - Querying the ledger for matching buy/sell orders

// - Matching orders based on price and quantity

// - Updating the ledger with the matched orders

// - Emitting an OrderMatchedEvent

// For demonstration purposes, let's emit a sample matched event

matchEvent := OrderMatchedEvent{

BuyOrderID: "buyOrder123",

SellOrderID: "sellOrder123",

MatchedQty: 10.0,

MatchedPrice: 100.0,

}

matchEventBytes, \_ := json.Marshal(matchEvent)

err := ctx.GetStub().SetEvent("OrderMatched", matchEventBytes)

if err != nil {

return err

}

return nil

}

// Main function and other functionalities remain unchanged

1. NFT\_supplychain

package main

import (

"encoding/json"

"errors"

"fmt"

"time"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

type SmartContract struct {

contractapi.Contract

}

type Package struct {

ID string `json:"id"`

DateOfManufacture string `json:"dateOfManufacture"` // ISO 8601 date format

PlaceOfOrigin string `json:"placeOfOrigin"` // "City, Country"

CurrentStatus string `json:"currentStatus"`

CustomsDetails []string `json:"customsDetails"`

IsDamaged bool `json:"isDamaged"`

}

type NFT struct {

ID string `json:"id"`

Owner string `json:"owner"`

Category string `json:"category"` // "Antibiotic", "Painkiller", etc.

}

// InitLedger - Initialize the ledger with default values

func (c \*SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {

// Initializations if needed

return nil

}

// CreatePackage - Create a new package

func (c \*SmartContract) CreatePackage(ctx contractapi.TransactionContextInterface, id, dateOfManufacture, placeOfOrigin string) error {

// Validate dateOfManufacture and placeOfOrigin

if \_, err := time.Parse(time.RFC3339, dateOfManufacture); err != nil {

return fmt.Errorf("invalid date format: %s", err.Error())

}

if placeOfOrigin == "" {

return errors.New("place of origin must not be empty")

}

pkg := Package{

ID: id,

DateOfManufacture: dateOfManufacture,

PlaceOfOrigin: placeOfOrigin,

CurrentStatus: "CREATED",

CustomsDetails: []string{},

IsDamaged: false,

}

packageBytes, \_ := json.Marshal(pkg)

return ctx.GetStub().PutState(id, packageBytes)

}

// UpdatePackageStatus - Update the status of a package

func (c \*SmartContract) UpdatePackageStatus(ctx contractapi.TransactionContextInterface, id, newStatus string) error {

// Retrieve the package from the ledger

packageBytes, err := ctx.GetStub().GetState(id)

if err != nil {

return fmt.Errorf("failed to get package: %s", err.Error())

}

if packageBytes == nil {

return fmt.Errorf("package not found: %s", id)

}

var pkg Package

err = json.Unmarshal(packageBytes, &pkg)

if err != nil {

return fmt.Errorf("failed to unmarshal package: %s", err.Error())

}

// Update the package status

pkg.CurrentStatus = newStatus

packageBytes, \_ = json.Marshal(pkg)

return ctx.GetStub().PutState(id, packageBytes)

}

// MintNFT - Create a new NFT for the package

func (c \*SmartContract) MintNFT(ctx contractapi.TransactionContextInterface, id, owner, category string) error {

// Validate category if needed

nft := NFT{

ID: id,

Owner: owner,

Category: category,

}

nftBytes, \_ := json.Marshal(nft)

return ctx.GetStub().PutState(id, nftBytes)

}

// TransferNFT - Transfer an NFT to a new owner

func (c \*SmartContract) TransferNFT(ctx contractapi.TransactionContextInterface, id, newOwner string) error {

nftBytes, \_ := ctx.GetStub().GetState(id)

if nftBytes == nil {

return fmt.Errorf("NFT not found")

}

var nft NFT

\_ = json.Unmarshal(nftBytes, &nft)

nft.Owner = newOwner

updatedNFTBytes, \_ := json.Marshal(nft)

return ctx.GetStub().PutState(id, updatedNFTBytes)

}

// GetNFT - Retrieve an NFT from the ledger

func (c \*SmartContract) GetNFT(ctx contractapi.TransactionContextInterface, id string) (NFT, error) {

nftBytes, \_ := ctx.GetStub().GetState(id)

if nftBytes == nil {

return NFT{}, fmt.Errorf("NFT not found")

}

var nft NFT

\_ = json.Unmarshal(nftBytes, &nft)

return nft, nil

}

func main() {

chaincode, err := contractapi.NewChaincode(new(SmartContract))

if err != nil {

fmt.Printf("Error create supply chain chaincode: %s", err.Error())

return

}

if err := chaincode.Start(); err != nil {

fmt.Printf("Error starting supply chain chaincode: %s", err.Error())

}

}

### **Testing Considerations:**

* Ensure that the package and NFT are correctly created with the provided details.
* Validate the transfer of the NFT across different entities in the supply chain.
* Check the updates to the package's status as it moves through the supply chain.
* Confirm the final ownership of the NFT (should be with Bob).

1. Mint event tickets

package main

import (

"encoding/json"

"fmt"

"strconv"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

type SmartContract struct {

contractapi.Contract

}

// Ticket, User, Event, and other struct definitions remain unchanged

func (t \*SmartContract) MintTicket(ctx contractapi.TransactionContextInterface, args []string) error {

if len(args) != 4 {

return fmt.Errorf("expected 4 arguments: TicketID, EventID, Seat, Owner")

}

ticket := Ticket{

TicketID: args[0],

EventID: args[1],

Seat: args[2],

Owner: args[3],

IsTraded: false,

IsAttended: false,

}

ticketAsBytes, \_ := json.Marshal(ticket)

return ctx.GetStub().PutState(ticket.TicketID, ticketAsBytes)

}

func (t \*SmartContract) BuyTicket(ctx contractapi.TransactionContextInterface, args []string) error {

if len(args) != 4 {

return fmt.Errorf("expected 4 arguments for buying a ticket")

}

ticketAsBytes, \_ := ctx.GetStub().GetState(args[0])

if ticketAsBytes == nil {

return fmt.Errorf("ticket not found")

}

var ticket Ticket

json.Unmarshal(ticketAsBytes, &ticket)

ticket.Owner = args[3]

ticket.IsTraded = true

ticketAsBytes, \_ = json.Marshal(ticket)

return ctx.GetStub().PutState(ticket.TicketID, ticketAsBytes)

}

func (t \*SmartContract) MarkAttendance(ctx contractapi.TransactionContextInterface, args []string) error {

if len(args) != 1 {

return fmt.Errorf("expected 1 argument: TicketID")

}

ticketID := args[0]

ticketAsBytes, \_ := ctx.GetStub().GetState(ticketID)

if ticketAsBytes == nil {

return fmt.Errorf("ticket not found.")

}

var ticket Ticket

json.Unmarshal(ticketAsBytes, &ticket)

ticket.IsAttended = true

ticketAsBytes, \_ = json.Marshal(ticket)

return ctx.GetStub().PutState(ticket.TicketID, ticketAsBytes)

}

func (t \*SmartContract) RegisterEvent(ctx contractapi.TransactionContextInterface, args []string) error {

if len(args) != 5 {

return fmt.Errorf("expected 5 arguments: EventID, EventName, Date, Location, IsSpecial")

}

isSpecial, err := strconv.ParseBool(args[4])

if err != nil {

return fmt.Errorf("invalid value for IsSpecial. Expected true or false.")

}

event := Event{

EventID: args[0],

EventName: args[1],

Date: args[2],

Location: args[3],

IsSpecial: isSpecial,

}

eventAsBytes, \_ := json.Marshal(event)

return ctx.GetStub().PutState(event.EventID, eventAsBytes)

}

// RedeemRewardPoints, TradeInTicketsForSpecial, and other functions as needed

func main() {

chaincode, err := contractapi.NewChaincode(new(SmartContract))

if err != nil {

fmt.Printf("Error creating chaincode: %s", err)

return

}

if err := chaincode.Start(); err != nil {

fmt.Printf("Error starting chaincode: %s", err)

}

}

### **Testing Considerations:**

* Ensure each function correctly updates the ledger state.
* Validate that tickets are minted, bought, and their ownership is transferred correctly.
* Check that event attendance marking works as expected.
* Monitor for errors or unexpected behavior in each transaction.

1. Second price sealed bid

Did not return expected result

package main

import (

"encoding/json"

"fmt"

"sort"

"strconv"

"time"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

type SmartContract struct {

contractapi.Contract

}

type AuctionItem struct {

ItemID string `json:"itemId"`

Name string `json:"name"`

Description string `json:"description"`

StartingBid float64 `json:"startingBid"`

EndTime time.Time `json:"endTime"`

Status string `json:"status"`

SellerID string `json:"sellerId"`

ReservePrice float64 `json:"reservePrice"`

}

type Bid struct {

ItemID string `json:"itemId"`

BidderID string `json:"bidderId"`

ActualBid float64 `json:"actualBid"`

SealedBid string `json:"sealedBid"`

AttachedValue float64 `json:"attachedValue"`

Timestamp time.Time `json:"timestamp"`

IsWithdrawn bool `json:"isWithdrawn"`

}

// StartAuction initializes a new auction

func (cc \*SmartContract) StartAuction(ctx contractapi.TransactionContextInterface, args []string) error {

// Example auction item details

item := AuctionItem{

ItemID: "AucItem123",

Name: "Vintage Watch",

Description: "A rare vintage watch from 1950",

StartingBid: 100.0,

EndTime: time.Now().Add(48 \* time.Hour), // 48 hours from now

Status: "ongoing",

SellerID: "Seller123",

ReservePrice: 200.0,

}

itemBytes, err := json.Marshal(item)

if err != nil {

return fmt.Errorf("error marshalling auction item: %s", err.Error())

}

return ctx.GetStub().PutState(item.ItemID, itemBytes)

}

// PlaceBid allows a bidder to place a bid on an auction item

func (cc \*SmartContract) PlaceBid(ctx contractapi.TransactionContextInterface, args []string) error {

// Example bid details

bid := Bid{

ItemID: "AucItem123",

BidderID: "Bidder123",

ActualBid: 150.0,

SealedBid: "encrypted-sealed-bid",

AttachedValue: 200.0,

Timestamp: time.Now(),

IsWithdrawn: false,

}

bidBytes, err := json.Marshal(bid)

if err != nil {

return fmt.Errorf("error marshalling bid: %s", err.Error())

}

return ctx.GetStub().PutState("BID\_" + bid.ItemID + "\_" + bid.BidderID, bidBytes)

}

// EndAuction closes an ongoing auction

func (cc \*SmartContract) EndAuction(ctx contractapi.TransactionContextInterface, args []string) error {

itemID := "AucItem123" // Example item ID

itemBytes, err := ctx.GetStub().GetState(itemID)

if err != nil {

return fmt.Errorf("error retrieving auction item: %s", err.Error())

}

var item AuctionItem

err = json.Unmarshal(itemBytes, &item)

if err != nil {

return fmt.Errorf("error unmarshalling auction item: %s", err.Error())

}

if time.Now().Before(item.EndTime) {

return fmt.Errorf("auction cannot be ended before its end time")

}

item.Status = "ended"

updatedItemBytes, \_ := json.Marshal(item)

return ctx.GetStub().PutState(item.ItemID, updatedItemBytes)

}

// RevealWinner calculates the winner of the auction

func (cc \*SmartContract) RevealWinner(ctx contractapi.TransactionContextInterface, args []string) (\*Bid, error) {

itemID := "AucItem123" // Example item ID

// Fetch all bids for the item and sort them by ActualBid in descending order

allBids := getAllBidsForItem(ctx, itemID)

sort.Slice(allBids, func(i, j int) bool {

return allBids[i].ActualBid > allBids[j].ActualBid

})

// Determine the winning bid based on Vickrey auction rules

if len(allBids) < 2 {

return nil, errors.New("not enough bids to determine a winner")

}

winningBid := allBids[0]

secondHighestBid := allBids[1]

// The winning price is the second-highest bid

winningBid.ActualBid = secondHighestBid.ActualBid

return &winningBid, nil

}

// ClaimExcessValue handles returning excess bid value to the winning bidder

func (cc \*SmartContract) ClaimExcessValue(ctx contractapi.TransactionContextInterface, args []string) error {

// Example item and bidder IDs

itemID := "AucItem123"

bidderID := "Bidder123"

winningBid, err := cc.RevealWinner(ctx, []string{itemID})

if err != nil {

return err

}

if winningBid.BidderID != bidderID {

return errors.New("only the winning bidder can claim excess value")

}

// Calculate excess value

excessValue := winningBid.AttachedValue - winningBid.ActualBid

fmt.Printf("Excess value of %f returned to bidder %s\n", excessValue, bidderID)

return nil

}

// getAllBidsForItem fetches all bids for a given item

func getAllBidsForItem(ctx contractapi.TransactionContextInterface, itemID string) []Bid {

// For simplicity, this example returns a hardcoded list of bids

return []Bid{

{ItemID: itemID, BidderID: "Bidder123", ActualBid: 150.0, AttachedValue: 200.0, Timestamp: time.Now()},

{ItemID: itemID, BidderID: "Bidder456", ActualBid: 250.0, AttachedValue: 300.0, Timestamp: time.Now()},

}

}

func main() {

chaincode, err := contractapi.NewChaincode(new(SmartContract))

if err != nil {

fmt.Printf("Error creating chaincode: %s", err)

return

}

if err := chaincode.Start(); err != nil {

fmt.Printf("Error starting chaincode: %s", err)

}

}

### **Testing the Chaincode:**

* **Start an Auction**: Invoke StartAuction to create a sample auction.
* **Place Bids**: Use PlaceBid to simulate bids from multiple bidders.
* **End the Auction**: Call EndAuction after the auction's end time.
* **Reveal the Winner**: Invoke RevealWinner to calculate the auction's winning bid.
* **Claim Excess Value**: Use ClaimExcessValue to handle the excess bid value return.

1. First price bid

Did not return expected result

package main

import (

"encoding/json"

"fmt"

"strconv"

"time"

"github.com/hyperledger/fabric-contract-api-go/contractapi"

)

type SmartContract struct {

contractapi.Contract

}

// Artwork, Bid, NFT, and Escrow struct definitions remain unchanged

func (s \*SmartContract) createArtwork(ctx contractapi.TransactionContextInterface, args []string) error {

// Args: ArtID, Title, Description, Artist, MinBid, ReservePrice, StartTime, BiddingPeriod, RevealPeriod

if len(args) != 9 {

return fmt.Errorf("incorrect number of arguments. expecting 9")

}

minBid, \_ := strconv.ParseFloat(args[4], 64)

reservePrice, \_ := strconv.ParseFloat(args[5], 64)

startTime, \_ := strconv.ParseInt(args[6], 10, 64)

biddingPeriod, \_ := strconv.ParseInt(args[7], 10, 64)

revealPeriod, \_ := strconv.ParseInt(args[8], 10, 64)

artwork := Artwork{

ArtID: args[0],

Title: args[1],

Description: args[2],

Artist: args[3],

MinBid: minBid,

ReservePrice: reservePrice,

StartTime: startTime,

BiddingPeriod: biddingPeriod,

RevealPeriod: revealPeriod,

AuctionEnd: startTime + biddingPeriod + revealPeriod,

HighestBid: 0,

HighestBidder: "",

Status: "open",

}

artworkAsBytes, \_ := json.Marshal(artwork)

return ctx.GetStub().PutState(artwork.ArtID, artworkAsBytes)

}

func (s \*SmartContract) placeBid(ctx contractapi.TransactionContextInterface, args []string) error {

// Args: BidID, ArtID, UserID, Amount

if len(args) != 4 {

return fmt.Errorf("incorrect number of arguments. expecting 4")

}

amount, \_ := strconv.ParseFloat(args[3], 64)

bid := Bid{

BidID: args[0],

ArtID: args[1],

UserID: args[2],

Amount: amount,

Timestamp: time.Now().Unix(),

}

// Validate the bid

// Retrieve the artwork from the ledger

// Check if the bid is higher than the current highest bid

// Update the artwork's highest bid and bidder

bidAsBytes, \_ := json.Marshal(bid)

return ctx.GetStub().PutState(bid.BidID, bidAsBytes)

}

func (s \*SmartContract) closeAuction(ctx contractapi.TransactionContextInterface, args []string) error {

// Args: ArtID

if len(args) != 1 {

return fmt.Errorf("incorrect number of arguments. expecting 1")

}

artID := args[0]

// Retrieve the artwork from the ledger

// Check if the current time is past the auction end time

// Change the artwork's status to "closed"

// Issue an NFT to the highest bidder

// Transfer the highest bid amount from the escrow to the artist

return nil

}

// Additional functions like getArtwork, revealBids, transferNFT, etc. remain unchanged

func main() {

chaincode, err := contractapi.NewChaincode(new(SmartContract))

if err != nil {

fmt.Printf("Error creating chaincode: %s", err)

return

}

if err := chaincode.Start(); err != nil {

fmt.Printf("Error starting chaincode: %s", err)

}

}

### **Testing Considerations:**

* Ensure the artwork is created correctly and stored on the ledger.
* Validate bid placement, including checking if the bids are higher than the current highest bid.
* Confirm the auction closure, including updating the artwork status and issuing an NFT to the highest bidder.
* Test the functionality of escrow management, including funds transfer and refund where applicable.