*Module 5*

# Exploring Smart Contracts in Hyperledger Fabric + Composer

IBM **Blockchain**



IBM

# What are Smart Contracts


An option contact between parties is written as code into the blockchain. The individuals involved are anonymous, but the contact is the public ledger.

A triggering event like an expiration date and strike price is hit and the contract executes itself according to the coded terms.

Regulators can use the blockchain to understand the activity in the market while maintaining the privacy of individual actors' positions

https://blockgeeks.com/guides/smart-contracts/

# ChainCode and Smart Contracts

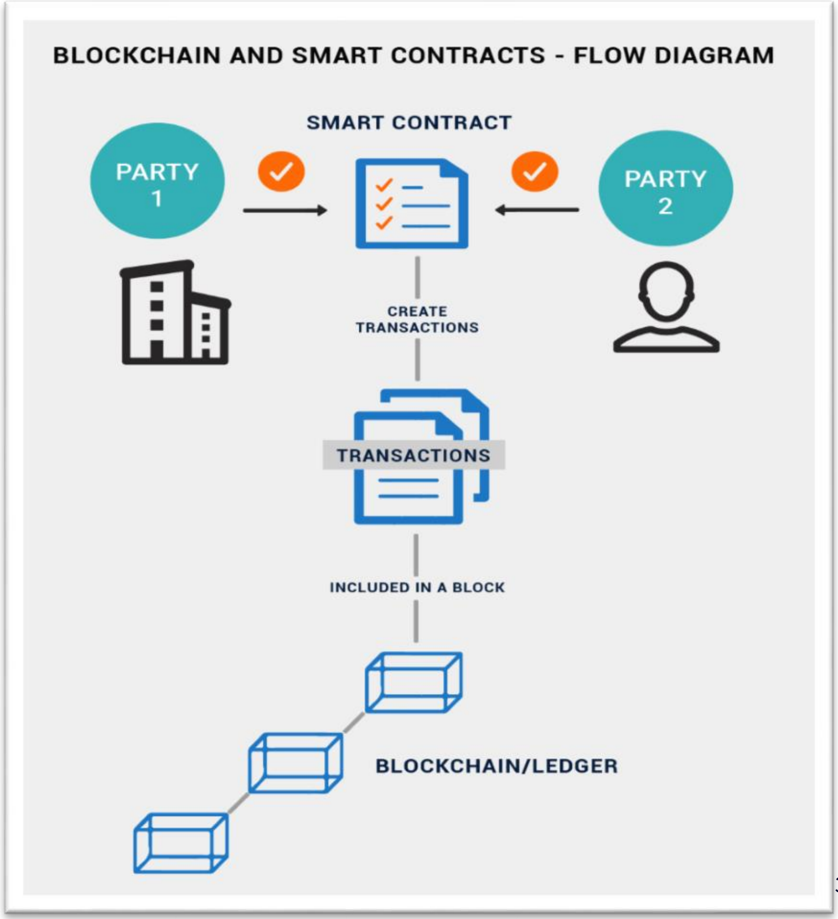


**Smart contracts** are simply ***computer programs*** that execute predefined actions when certain conditions within the system are met.

Smart contracts are implemented using **chaincode** which provides the **language of transactions** that allow the ledger state to be modified. This can represent the exchange and transfer of anything (e.g. shares, money, content, property).

Chaincode can be written in:

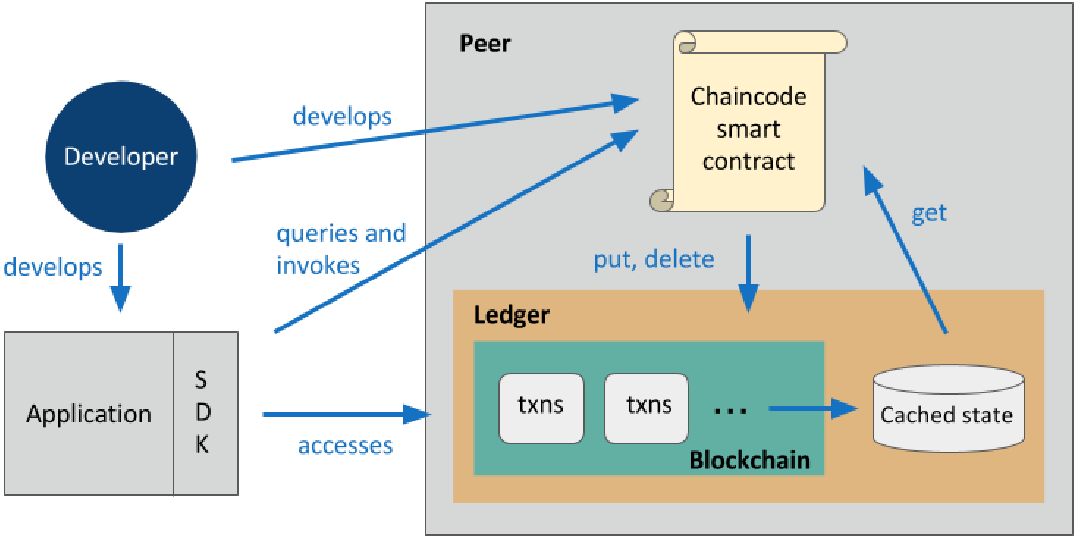
- Go
- Javascript (Hyperledger Composer)
- Java (less popular)



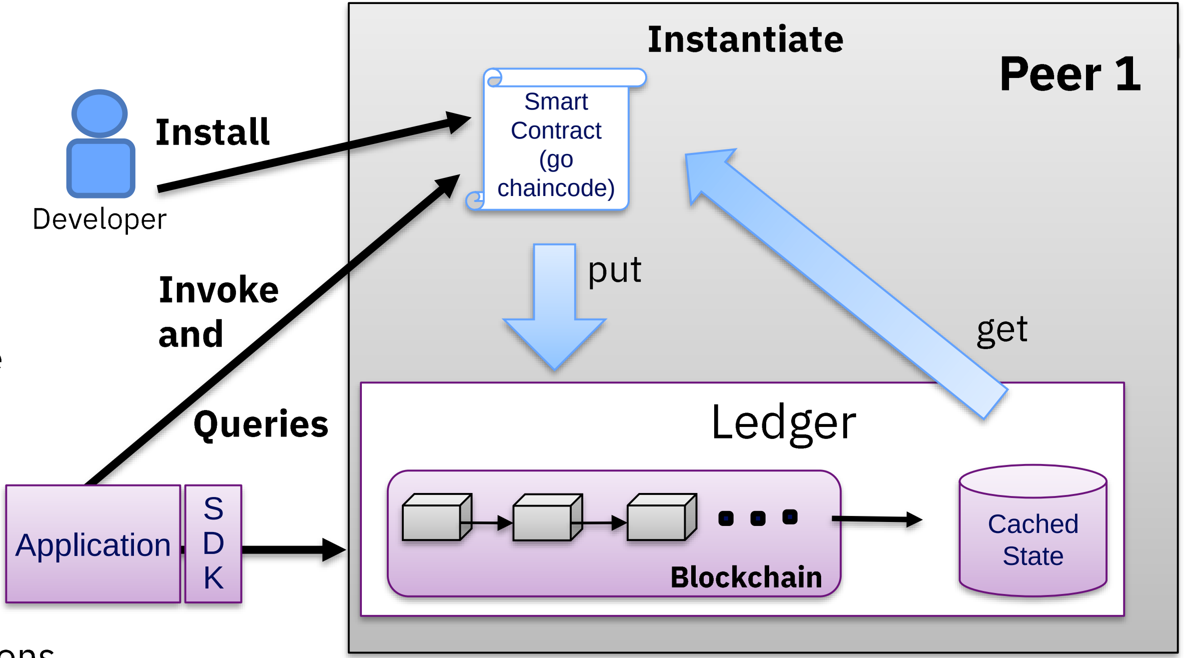
# Blockchain App Basic Flow



1. The **Developer** writes the **Blockchain application** using the **Go language** to implement the chaincode
2. The chaincode must to be **installed** on every peer that will endorse a transaction and instantiated on the channel.
3. The **Blockchain app** will invoke the smart contract via the **Hyperledger Fabric Client SDK.**
4. These calls are processed by the business logic within the chaincode smart contract.
5. A `put` or `delete` command will go through the consensus process and will be added to the blockchain within the ledger.
6. A `get` command can only read from the world state, but **it is not recorded** on the blockchain.
7. Depending on the implementation of the world state DB, rich queries can also be used to retrieve ledger data



# Chaincode lifecycle




1. **Developer writes chaincode**
   Developers implement the business logic
2. **Chaincode is installed**
   The process of placing a chaincode on a peer's file system
3. **Chaincode is instantiated**
   Starts and initializes chaincode on a specific channel. Once complete peers have chaincode that can accept chaincode invocations
4. **Execute**
   The current ledger state represents the latest values for all keys **(World State),** known to the channel. Chaincode executes transaction proposals against current state data
5. **Query**
   Reads the current state of ledger but does not write to the ledger. **Queries** typically don't submit these read-only transactions for ordering, validation, and commit.

# Structure of a Blockchain App

Here you can see a typical file structure of the Fabric application:

```
▼ chaincode
    ▼ tuna-app
        tuna-chaincode.go
▼ tuna-app
    ▼ client
        app.js
        favicon.png
        index.html
    ▼ src
        changeTunaHolder.js
        queryAllTuna.js
        queryTuna.js
        recordTuna.js
    .gitignore
    controller.js
    package-lock.json
    package.json
    README.md
    registerAdmin.js
    registerUser.js
    routes.js
    server.js
    startFabric.sh
```

This is the chaincode file that contains all our business logic for our sample tuna fish example application. This is deployed on the network peers and from our application's backend we use the SDK to call functions within this smart contract.

Javascript client code in *app.js* manipulates html elements for the user interface.

HTML file containing the UI for our client application. You will view this file in your browser to begin interfacing with Hyperledger Fabric.

Folder containing code that uses SDK to connect client requests to network and chaincode functions.

*registerAdmin.js* assigns an admin user to the fabric client and *registerUser.js* registers and enrolls user to interact with the fabric network

*routes.js* sends requests (and any information encoded in request URLs) to the controller functions. *controller.js* contains functions that perform operations and interrogate data.

The bulk of our application backend logic is in this file. When we start our application locally we will run **`node server.js`** to view our UI at **`localhost:8000`**

Contains all the necessary commands to start our simple Hyperledger Fabric network which we will do at the beginning with: **`./startFabric.sh`**

# GO chaincode structure



Chaincode must be in the main package, with a (main() function) and implement the Init and Invoke methods

The import statement lists a few dependencies that you will need for your chaincode to build successfully.

fmt – contains Println for debugging/logging.

errors – standard go error format.

github.com/hyperledger/fabric/core/chaincode/shim – contains the definition for the chaincode interface and the chaincode stub, which you will need to interact with the ledger.

```go
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "strings"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
)

type SimpleChaincode struct {}

type User struct {
    ID        string `json:"id"`
    LegalName string `json:"legalName"`
    Role      string `json:"role"`
    UserCert  string `json:"certificate"`
}

func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    var err error
    fmt.Printf("\n\nInitialization successful")
    return shim.Success(nil)
}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    function, args := stub.GetFunctionAndParameters()
    fmt.Printf("\n\n\n###### INVOKE: function: %v, args: %v\n", function, args)
    if function == "registerUser" {
        return t.registerUser(stub, args)
    } else if function == "doSomethingReallyCool" {
        return t.doSomethingReallyCool(stub, args)
    }
    return shim.Error("Unknown function: " + function)
}

func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}
```

# Chaincode Key APIs



The ChaincodeStub provides functions that allow you to interact with the underlying ledger to query, update, and delete assets. The key APIs for chaincode include:

**func (stub *ChaincodeStub) GetState(key string) ([]byte, error)**
Returns the value of the specified key from the ledger. Note that **GetState** doesn't read data from the Write set, which has not been committed to the ledger. In other words, **GetState** doesn't consider data modified by **PutState** that has not been committed. If the key does not exist in the state database, **(nil, nil)** is returned.

**func (stub *ChaincodeStub) PutState(key string, value []byte) error**
Puts the specified key and value into the transaction's Write set as a data-write proposal. **PutState** doesn't affect the ledger until the transaction is validated and successfully committed.

**func (stub *ChaincodeStub) DelState(key string) error**
Records the specified key to be deleted in the Write set of the transaction proposal. The key and its value will be deleted from the ledger when the transaction is validated and successfully committed.

# Transaction Processor Functions

IBM Blockchain

When using Hyperledger Composer, Smart Contracts are written in JavaScript, and are implemented as **Transaction Processor Functions**

Decorators within documentation comments are used to annotate the functions with metadata required for runtime processing.

Transaction processor functions are automatically invoked by the runtime when transactions are submitted using the BusinessNetworkConnection API.

```
/**
 * Sample transaction processor function.
 * @param {org.acme.sample.SampleTransaction} tx The sample transaction instar
 * @transaction
 */
function sampleTransaction(tx) {
    // The relationships in the transaction are automatically resolved.
    // This means that the asset can be accessed in the transaction instance.
    var asset = tx.asset;
    // The relationships are fully or recursively resolved, so you can also
    // access nested relationships. This means that you can also access the
    // owner of the asset.
    var owner = tx.asset.owner;
}
```

# Working with registries

IBM Blockchain

The state of assets are stored in asset registries, which are queried and updated within transaction processor functions

Composer defines JavaScript APIs and REST APIs to submit transactions and to create, retrieve, update, and delete assets within asset registries

```
/**
 * Sample transaction processor function.
 * @param {org.acme.sample.SampleTransaction} tx The sample transaction instanc
 * @transaction
 */
function sampleTransaction(tx) {
    // Update the value in the asset.
    var asset = tx.asset;
    asset.value = tx.newValue;
    // Get the asset registry that stores the assets. Note that
    // getAssetRegistry() returns a promise, so we have to return
    // the promise so that Composer waits for it to be resolved.
    return getAssetRegistry('org.acme.sample.SampleAsset')
        .then(function (assetRegistry) {
            // Update the asset in the asset registry. Again, note
            // that update() returns a promise, so so we have to return
            // the promise so that Composer waits for it to be resolved.
            return assetRegistry.update(asset);
        })
}
```

# Thank you

IBM **Blockchain**

www.ibm.com/blockchain

developer.ibm.com/blockchain

www.hyperledger.org



IBM