HYPERLEDGER

Developing on Hyperledger Fabric 1.2

Table of Contents

CHAPTER 1: INTRODUCTION TO BLOCKCHAIN

This guide describes what blockchain technology is, the need for blockchain and where it is used. It describes Hyperledger and Hyperledger Fabric which is used for developing blockchain applications and solutions with a modular architecture. Hyperledger Composer which creates the Hyperledger “Fabric” application is discussed in detail.

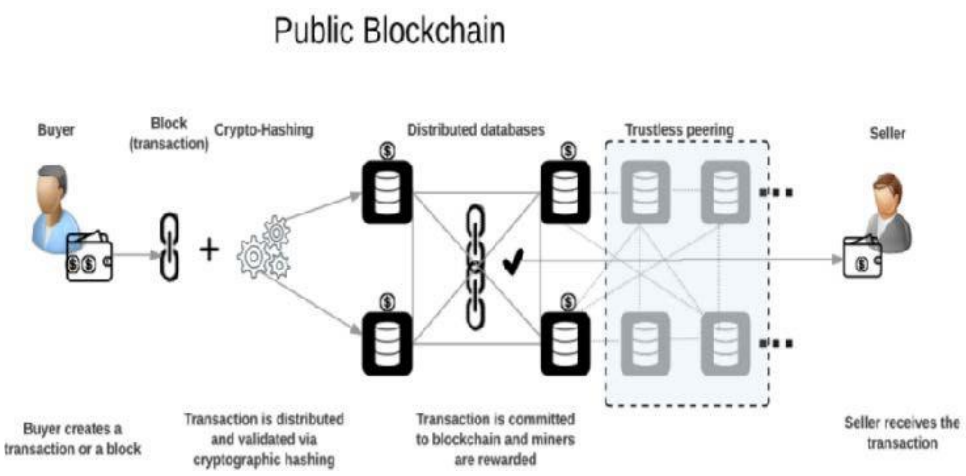
What Is blockchain

Blockchain can be described as a growing list of blocks which are linked to each other and secured by cryptography. Blockchain serves as an open distributed ledger that is replicated thousands of times in a given network. The database in a network is shared by millions of computers in a network simultaneously. The links are chained together by using hashing function.

Type of Networks in Blockchain

Public Network

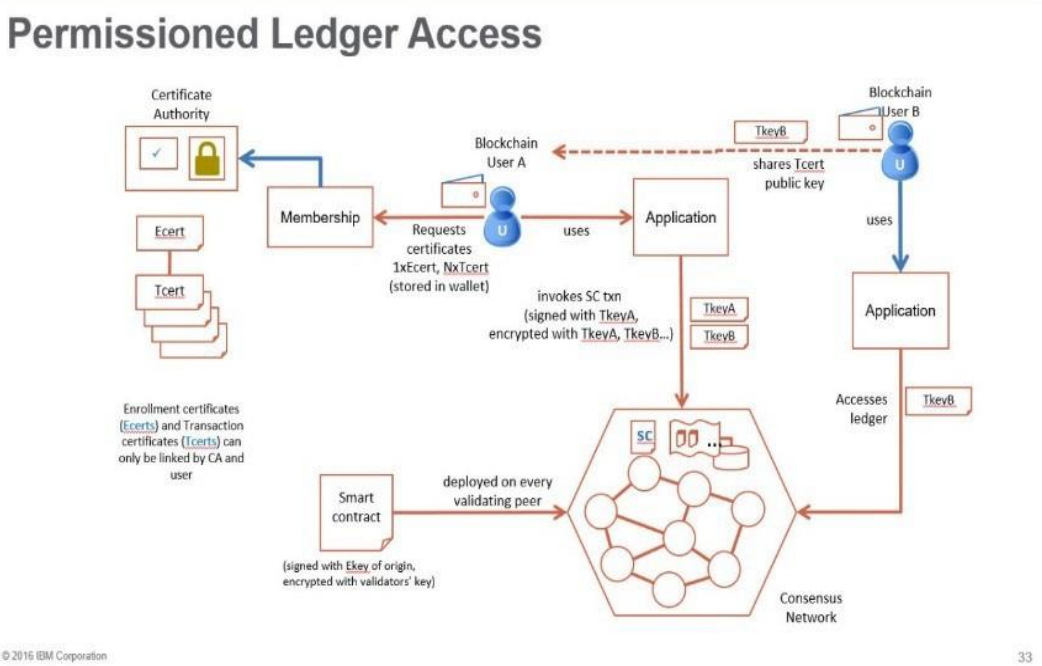
These are large and decentralized system of networks where anyone can participate. They are expensive to maintain and are slow. They are however more secure and immutable than private and permissioned networks. Bitcoin and Ethereum are examples of public networks.



Permissioned Network

On a permissioned network, only authorized persons or organizations can write data onto the blockchain. The organization authorizes people who can access and view the data. This type of network is less expensive and easier to maintain. They are very fast and require less storage space. The protocols and governance of these networks is authorized by an industry consortium.

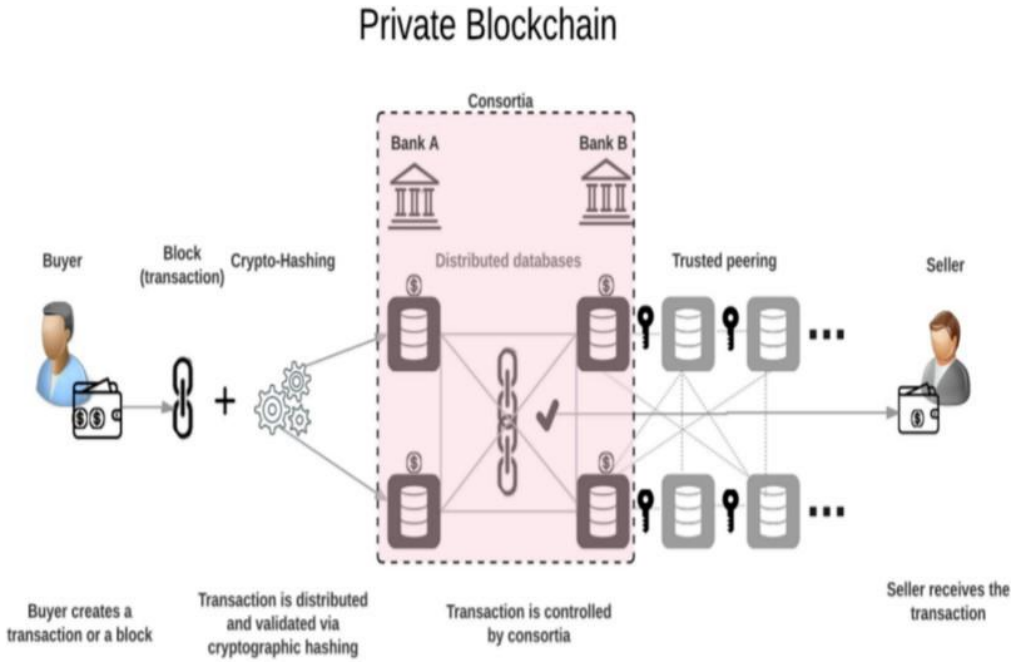
Example of permissioned network is Ripple.



Private network

In a private network, write permissions are kept centralized to one organization. Read permissions may be public or restricted to an arbitrary extent. The transactions made in a private network are faster, and a smaller transaction fee is required. It is a highly trusted network. The devices are well connected, and any faults can be resolved by human intervention, which can be easily approved by the users since the users trust the single organization in control of the blockchain.

Example of private network is Hyperledger Fabric.

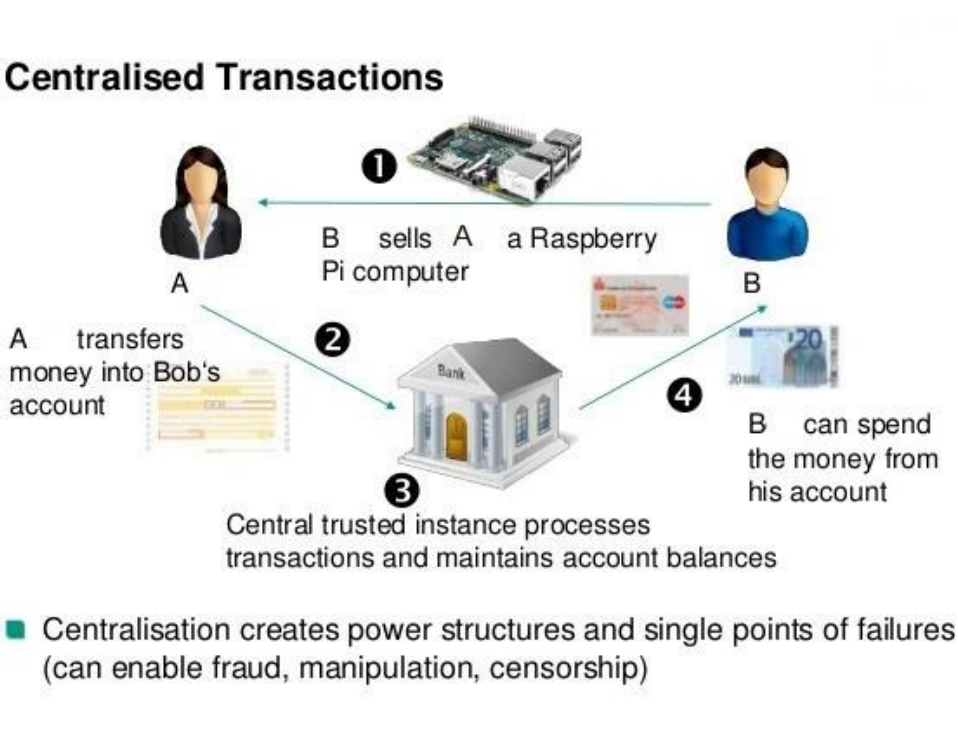


The Need for Blockchain

Centralized System

They are many participants involved in a business network. Participants can be a government organization, customer, suppliers, banks etc. The flow of goods, services and transaction are managed in a business network.

In a network of many organizational partners every participant has their own copy of the database. Each network manages their own centralized database, and each have their own set of protocols governing their database.

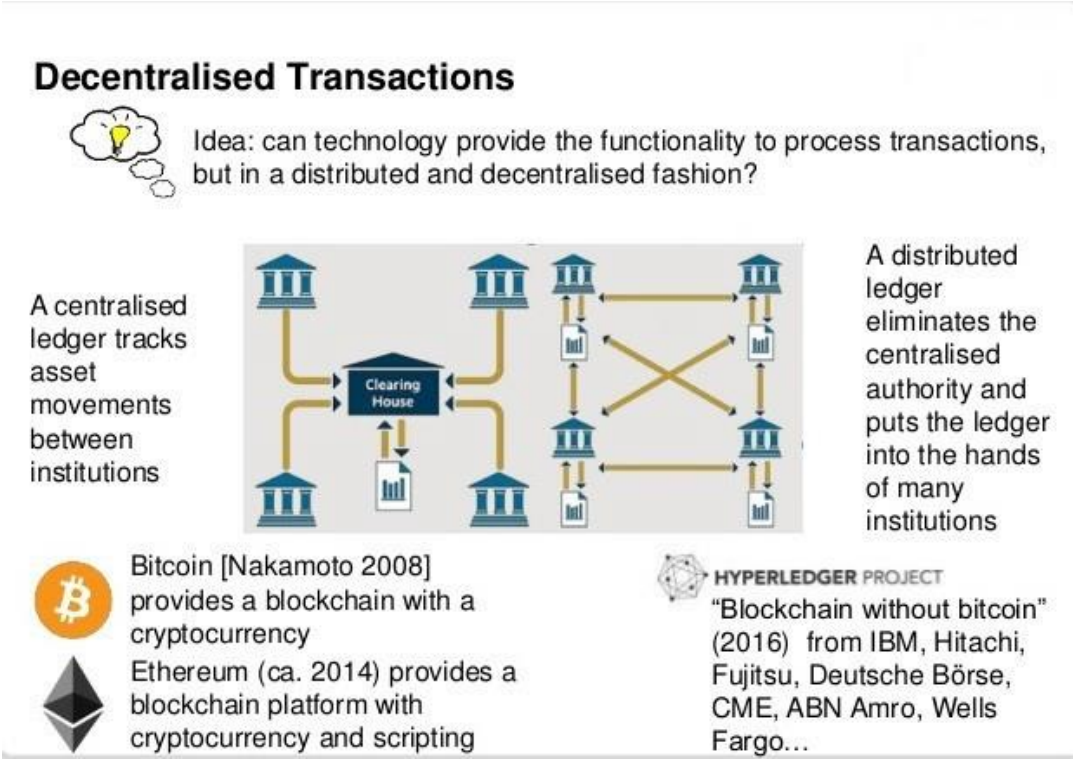


Disadvantages

- Maintaining individual databases requires a lot of time and effort. Also, a failure in one network results in the failure of the whole succeeding network.
- A lot of time is wasted in bringing up the network and continuing with the process of transfer.
- This system of centralized distributed system is very inefficient and expensive to maintain.
- It is vulnerable to fraud, malicious and cyber-attacks.

Decentralized System

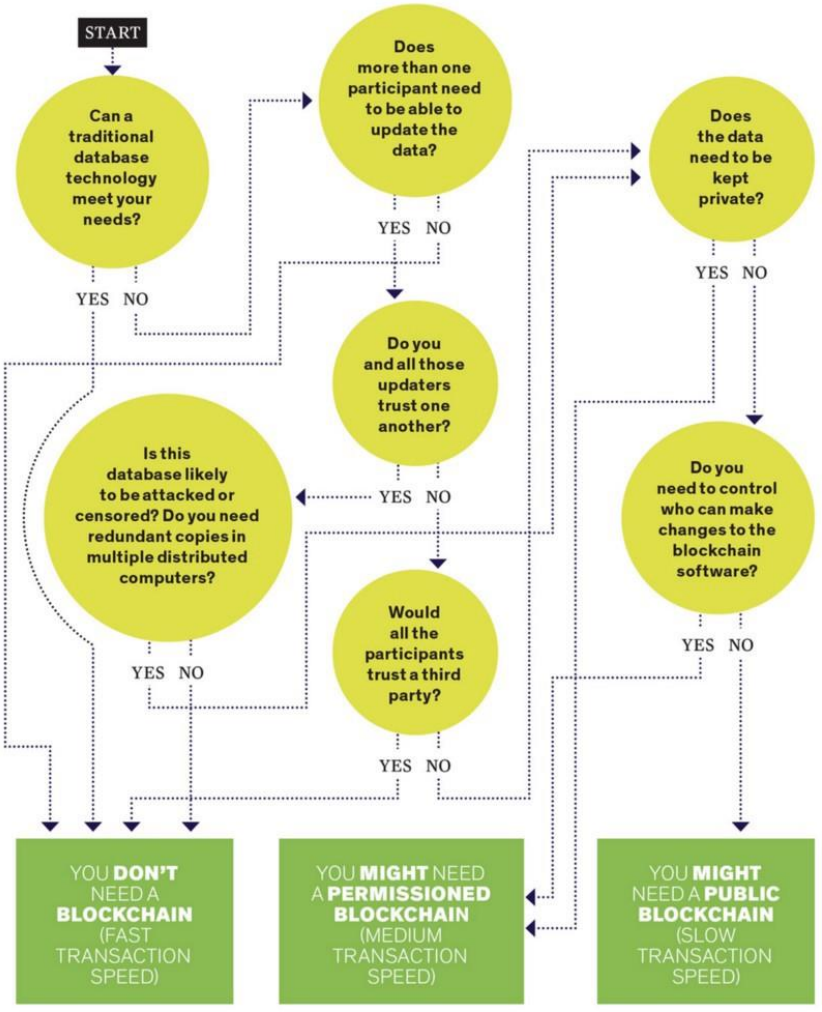
Blockchain solves the above-mentioned problem. Blockchain is a decentralized and distributed ledger that is used to record transactions across the network. By storing data across its network, the blockchain eliminates the risks that come with data being held centrally.



Advantages

- Each participant in a network has their own replicated copy of the ledgers. There is transparency to all the participants involved in the network.
- Anyone in the network can update the records with the help of consensus.
- It helps save time because any changes made are real time.
- The decentralized distributed system of records reduces cost and effort by cutting down intermediaries involved in the process of auditing.
- As everyone in the given network has the replicated copy of ledgers, a failure in one node does not affect the entire network.
- Blockchain networks are highly secure because it's a distributed system and is highly secured with a privacy system. It creates a digital trust among two participants as every system uses the same data and same sets of protocols for the flow of business processes.

Deciding to use a Blockchain. i.e. Who Needs One?



Components of Blockchain

The components that describe the blockchain are:

Consensus

Consensus breaks away from the older design of a centralized system. In a centralized system it is directly operated by known and trustworthy individuals of the organization. Blockchain on the other hand is an open decentralized database and controlled by many individuals. To create trust, laws are enforced which decides authority and trust in the network. This authority enables nodes to continuously and sequentially record transactions on a public book, creating a chain of blocks.

Every participant in a business network can submit information to the blockchain. The transaction can be mistrusting and malicious. Through consensus, the data is evaluated and agreed by all the participants of the network. After consensus is made the transaction gets permanently incorporated in the blockchain. This helps maintain consistency and trust that the transaction committed is valid. Every blockchain has they own algorithm for reaching consensus.

Provenance

Provenance means recording the history of data, from its inception to various stages of the data lifecycle. Provenance provides a detailed record of how the data was collected, where it was stored and how it is used. Blockchain holds complete provenance details of each component of data transfer. It is accessible to all the participants in a business network. It improves the system utilization and increases trust. It maintains a complex system of record in supply chain based industries.

Immutability

Immutability in blockchain means once a transaction has been written to the ledger it cannot be changed. This provides trust that once the data is written it cannot be altered. Blockchains are open distributed databases and are vulnerable to attack and tampering of data. To prevent this blockchain uses hashes and blocks. Hashes are the basis of security and immutability in a blockchain by using hashes that can easily detect whether the data is being tampered with.

Finality

Finality is same as immutability. This means that once a transaction is committed it can never be reverted back and changed.

Use cases Where Blockchain Is Being Used

Just a few of the industries that are developing and implementing blockchain are listed below. Many more are in the early developmental stages.

Banking Industries

Banking industries were first to recognize the potential of blockchain technology. The banking industry is highly regulated, and the cost of maintenance is very high. Blockchain technology can be used to decrease the cost of these transfers by reducing the need for banks to manually settle transactions, including international payment, capital markets, trade finance, etc.

Business/Manufacturing Industries

Business industries utilizing supply chain management are using blockchain for keeping track of goods as they move from one place to another. Blockchain's immutable property makes it suitable for keeping the track of goods. Blockchain provides a new and dynamic means of organizing tracking data and putting it to use on a worldwide basis.

Healthcare

Health industry has been identified as another major industry that can benefit from blockchain. Health data that's suitable for blockchain would include general information like age, gender, and potentially basic medical history data like immunization history or vital signs.

Real Estate

The real estate industry is one of the industries which will be largely affected by the innovation of blockchain technology. It would expedite home sales by quickly verifying finances, would reduce fraud thanks to its encryption, and would offer transparency throughout the entire selling and purchasing process.

Government Agencies

There are various applications of blockchain being researched currently that can support government functions and take the current model of e- government to the next level. The various areas of government agencies where blockchain application can be used are record management, identity management, voting, taxes, and non-profit agencies.

Cyber Security

The advantage of blockchain in cybersecurity is that it removes the risk of a single point of failure. Blockchain technology also provides end-to-end encryption and privacy.

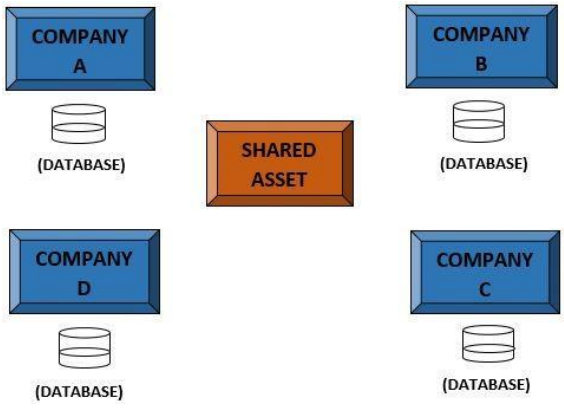
Other Industries

The other industries where blockchain can be used are financial management, shareholder voting, record management, big data, data storage, internet of things.

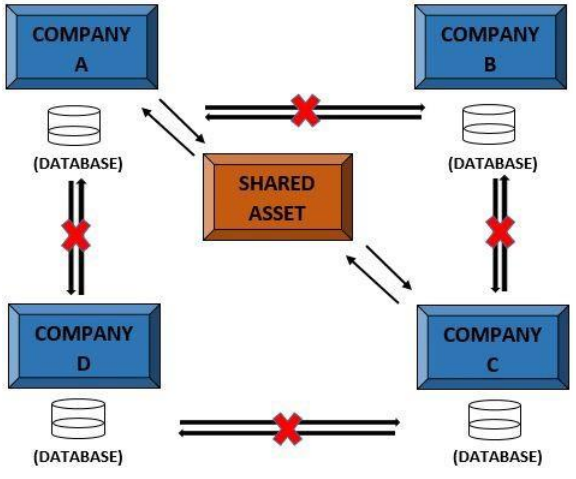
Asset Transfer Example on Blockchain

An example of asset transfer.

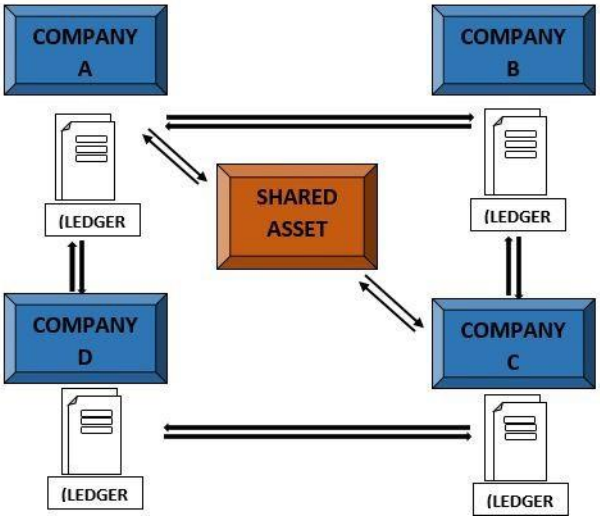
There are four companies in a business network and all are sharing a shared resource a container.



Company A has the container initially. Company C requests for the container. The container gets transferred from Company A to Company C. Once the container is transferred both A & C update they individual databases. Company B and C database do not get updated as all the participants have centralized system of database. Each company keep they own centralized database. Now, Company B requests for that asset. The container either has to be returned back to location A or location A has to inform location B that the asset has been transferred to location C. This consumes a lot of time and effort. It also creates a lot of ambiguity and is expensive to maintain.



To overcome this problem blockchain is used. In blockchain, individual nodes have the same replicated copy of the ledger. Whenever a transaction takes place all the ledgers in the network gets updated simultaneously. Thus, all the other nodes in the network can find out the transactional history of the container. Blockchain ensures provenance and immutability. It builds a digital trust among all the nodes in a network.



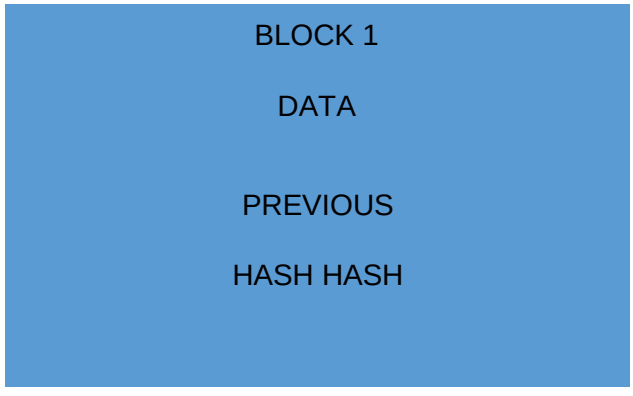
CHAPTER 2: HOW BLOCKCHAIN WORKS

Blockchains are intended to securely exchange assets without the involvement of middlemen. Blockchain data is served by decentralized distributed databases which work in cohesion, to record a consensus of a transaction and store a collection of transactions into a block. Blockchain uses cryptographic hashing to maintain the integrity of the data. Though creating hashes is trivial for a CPU, blockchain requires the hash values to have a specific form to get committed. This content hash attached to each block makes it immune to any further tampering of data, thus making the block 'immutable'. Generating a specific hash requires many nodes working together in full throttle. With each successful transaction to the ledger, participating nodes will be rewarded.

Structure of Blockchain

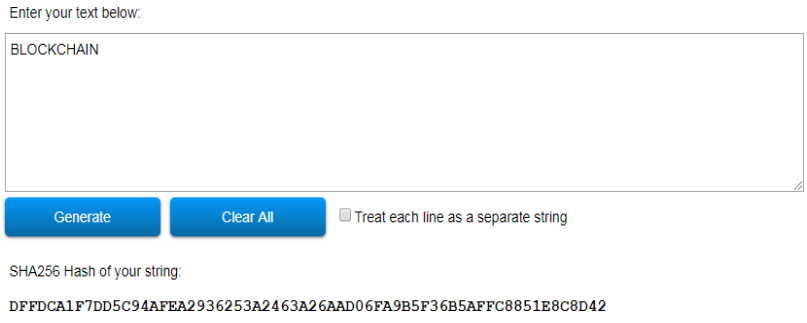
Block

A block contains data of the transaction, hash of the block and hash of the previous block. The structure of the block is

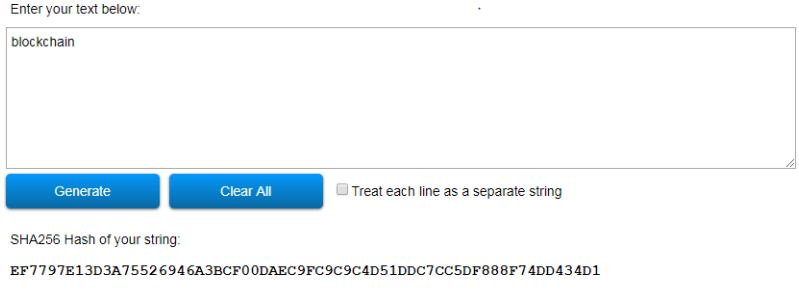


Hash

Hashing means taking an input string of any length and giving out an output of a fixed length. In the context of cryptocurrencies like Bitcoin, the transactions are taken as an input and run through a hashing algorithm which gives an output of a fixed length.



Above is an example of SHA256 Hash generation. No matter how big or small your input is, the output will always have a fixed 256-bits length.



Even if slight changes are made to the input the changes get reflected in the hash. Blockchain hash functions makes it immutable.

Blockchain

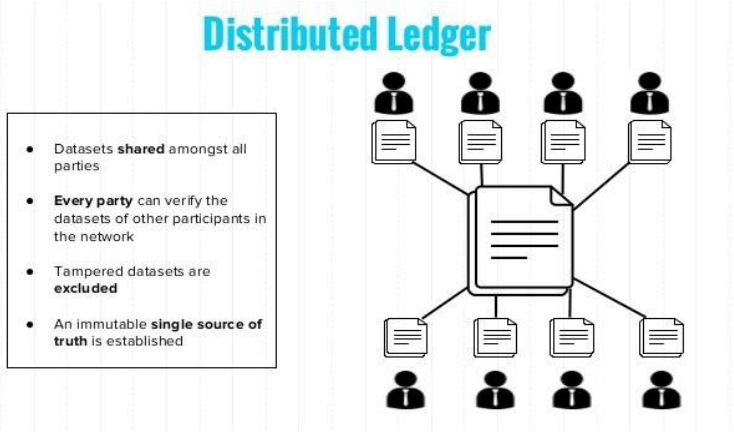
Blockchain consist of blocks chained together using a hash. Blockchain is immutable. Immutability is achieved by using the hash functionality. Blockchains are secure by design as every block contains the hash of the previous block. Any changes made are reflected throughout the chain.

BLOCK 1
DATA
PREVIOUS HASH
HASH

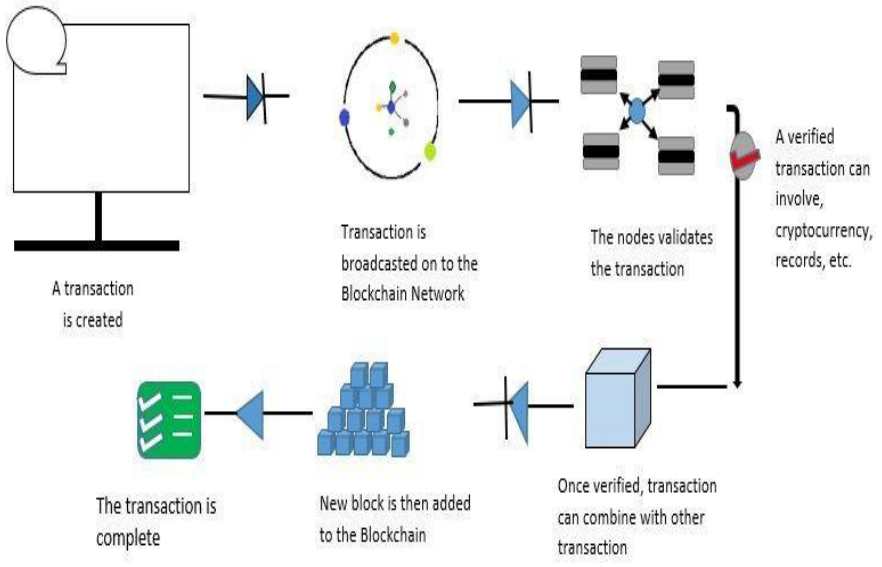
BLOCK 2
DATA
PREVIOUS HASH
HASH

Distributed

The blockchain database is distributed. Every node in the network have they own copy of the replicated ledgers. This makes the database tamper proof.



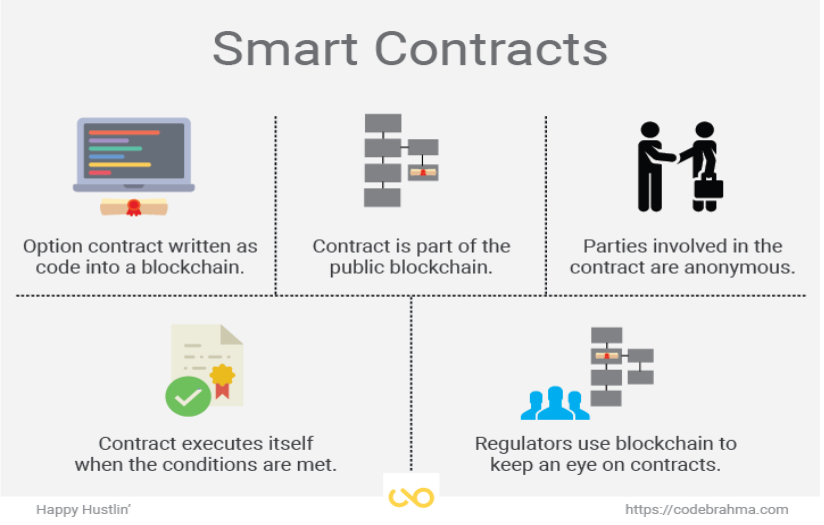
Life Cycle of a Blockchain



Smart Contract

Smart contracts are business logics that must be implemented in the blockchain. Smart contracts not only define the rules and penalties around an agreement in the same way that a traditional contract does, but also automatically enforces those obligations. Smart contracts must run multiple times to achieve consensus. It guarantees consistency in a transaction and detects duplicates in a blockchain. Smart contracts, often created by computer programmers through the help of smart contract development tools, are entirely digital and written using programming code languages such as C++, Go, Python, Java. This code defines the rules and consequences in the same way that a traditional legal document would, stating the obligations, benefits and penalties which may be due to either party in various circumstances. This code can then be automatically executed by a distributed ledger system.

Smart contracts provide autonomy in making an agreement. They also build trust as the documents are encrypted in a shared ledger. They also back up all data as it is duplicated many times, and transaction costs are reduced since the involvement of intermediaries are removed.



Consensus Algorithm

Consensus algorithm is agreement among a group of participants in a network. Consensus in a distributed system is challenging. Consensus algorithms are resilient to failures of nodes, partitioning of the network, message delays, messages reaching out-of-order and corrupted messages. They also must deal with selfish and deliberately malicious nodes.
Several algorithms are proposed in the research literature to solve this, with each algorithm making the required set of assumptions in terms of synchrony, message broadcasts, failures, malicious nodes, performance and security of the messages exchanged.

Different Types of Consensus Algorithms

Proof of Work

Proof of work is the most commonly used consensus algorithm. This algorithm is used by the largest cryptocurrency, Bitcoin. In proof of work, every participant in a network does not must send messages to reach consensus. The miners compete to add the next block (a set of transactions) in the chain by racing to solve an extremely difficult cryptographic puzzle. The miner who solves the puzzle first wins the mining fee that is paid. This is called proof of work meaning trying to prove that series of transaction are valid. It requires a lot of computational power and resources which makes it expensive.

Proof of Stake

In proof of stake the mining is done by a validator who is selected by the network based on stake. A validator is a person who has the highest amount of wealth(coins). This is efficient as there is less computational cost. No block rewards, only a transactional fee is rewarded. It is safer as network attacks become more expensive. Ethereum is now using proof of stake consensus.

Practical Byzantine Fault Tolerance

In Byzantine General Problem, Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching an agreement. The generals must decide on when to attack the city, but they need a strong majority of their army to attack at the same time. The generals must have an algorithm to guarantee that (a) all loyal generals decide upon the same plan of action, and (b) a small number of traitors cannot cause the loyal generals to adopt a bad plan. The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition (a) regardless of what the traitors do. The loyal generals should not only reach an agreement but should agree upon a reasonable plan.

The above problem is solved by Practical Byzantine General's Problem. IBM backed Hyperledger uses this consensus algorithm. In PBFT each node maintains an internal storage. When a node receives a message, it is signed by the node to verify its format. Once enough of the same responses are reached, then a consensus is met that the message is a valid transaction.

Separate Functions (actors) of Blockchain

Blockchain Developer

The blockchain developer's primary responsibility is to develop applications and smart contracts, and design how they interact with the ledger and the other components of the blockchain network.

Blockchain Operator

Blockchain operators are responsible for deployment and operations part of the blockchain.

Blockchain Regulator

Regulators have the overall authority in a business network. They have high privilege to access the ledgers content. Blockchain architects have broad understanding of applications, smart contracts, events and integration, peers, consensus and security. They must keep consideration about performance, maintenance, security and functioning of the network.

Blockchain User

The blockchain users are the participants in a network. They interact with the blockchain without knowing the internal functionality of the blockchain.

Membership Service Provider

The Membership Service Provider manages the different type of certificates needed for permissions in the network.

CHAPTER 3: INTRODUCTION TO HYPERLEDGER

What is Hyperledger

Hyperledger is an open source collaborative effort created for open industrial blockchain development. It started in December 2015 by the Linux Foundation. Linux Foundation's objectives were to create an environment in which communities of software developers and companies meet and coordinate to build blockchain frameworks.

Why Hyperledger

Although blockchain has emerged as a world changing industry, it is challenged by the lack of scalability, and the lack of support for confidential and private transactions. To meet the varied demands of the critical and complex business market demands, Hyperledger has been designed and implemented to resolve these issues.

Each blockchain network serves a different goal with each network ledger. Every ledger has its own core functionality, and one ledger is not dependent on other's network ledger functionality. Although every network is independent of each other, it still must address each other to allow transactions on one ledger, allowing it to discover and utilize the appropriate transaction and smart contracts on the other ledgers.

The Hyperledger blockchain platforms are (but not limited to):

Hyperledger Burrow

Hyperledger Burrow is a blockchain client which includes a built-in specification for the Ethereum Virtual Machine. Contributed by Monax and Intel.

Hyperledger Fabric

Fabric is a permissioned blockchain infrastructure that provides plug and play implementation of blockchain. Contributed by IBM.

Hyperledger Iroha

Iroha was contributed by a group of Japanese companies to incorporate blockchain framework, specifically designed for mobile applications.

Hyperledger Sawtooth

Hyperledger Sawtooth is a modular platform for building, deploying, and running distributed ledgers. Distributed ledgers provide a digital record (such as asset ownership) that is maintained without a central authority or implementation. It uses consensus algorithm mechanism known as "Proof of elapsed time". Proof of Elapsed Time is designed to create a consensus model which mainly focuses on efficiency. It mainly focuses on fairness, investment and verification.

Hyperledger Application

Business contracts can be coded into smart contracts to allow two parties to automate contractual agreement with high degree of trust. Hyperledger provides privacy control to protect sensitive business information from being disclosed to outside parties which may also have access to ledger.

They are also instances where contracts must made public, accessible to all parties of a ledger. Example is a ledger used for offers seeking bids. This type of ledger is standardized so that the bidders can easily find them.

Assets like Financial securities must be able to be dematerialized on a blockchain network. The stake holder of the asset type has direct access to each asset, allowing them to initiate trade and acquire information of an asset without going through layers of intermediaries. A stakeholder must be able to access asset information in real time and should be able to add business rules for any given asset type, which reduces cost by implementing automation logic. The creator of an asset can make the asset private and confidential, or public as the use case warrants.

The blockchain fabric provides a means to every participant on a supply chain network to input and track the details of asset transfer right from production to storage to final sale of goods and services. It helps track provenance and maintain immutable records of all aspects of the product lifecycle. It helps keep provenance, so it can deep search backwards in time through many transactional layers.

Hyperledger Architecture

The Hyperledger is divided into three categories; Membership, Blockchain, and Chaincode.

Membership Services

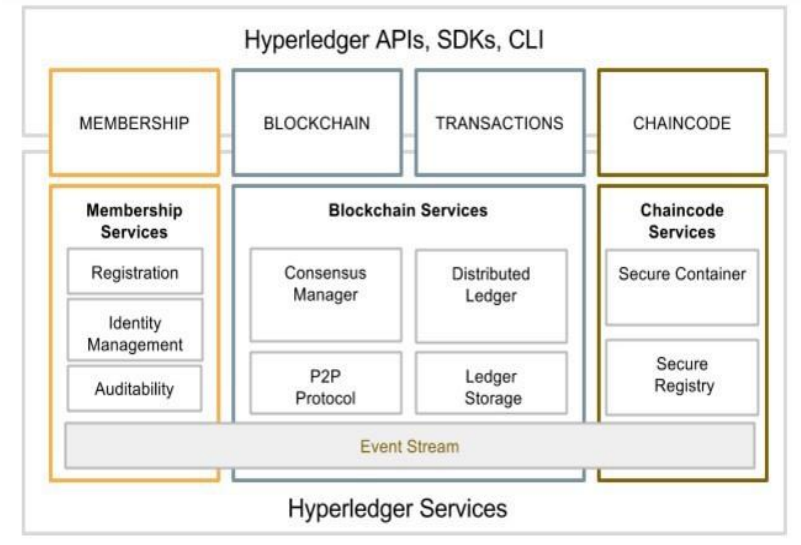
Membership services provides identity, privacy, and confidentiality to a network. For transacting the participants must obtain identities. The Reputation Manager enables auditors to view transactions pertaining to a participant, providing that each auditor has been granted proper access authority by the participants.

Blockchain

Blockchain services manages the distributed ledger through a peer to peer protocol that is built on HTTP/2. The optimized data structure provides efficient schemes for maintaining the world state replicated at many participants. Different consensus algorithms are enforced to guarantee strong consistency.

Chaincode

Chaincode services are secured and lightweight. The environment is a “locked down” and secured container with a set of signed base images which contains secure OS and Chaincode language, runtime and SDK images for Golang, Java, and Node.js. Additional programming language can also be used if required.



Membership

Hyperledger is a private validator network protocol. All the entities in a network must register with membership services to obtain an identity with access and transaction authority on the network. Validators determine the level of permissions required to transact. The network setup also defines the network as permissive, allowing the ease of access. It supports for rapid and high adoption for a more controlled and restrictive environment.

Registration services

Registration service provides the control and management of authorizations for Hyperledger participation.

Identity Management

Identity Management services offers assurance and authorized disclosure of association of identities and roles to Hyperledger participants.

Auditability Services

Auditability services provides the capability to provide authorized entities the means to link transactions of individual users or groups of users according to the affiliation or roles. It also provides capability to access the system activity of an individual user, or the system itself.

Blockchain's Main Components

Blockchain consists of three major components; P2P Protocol, Distributed Ledger and Consensus Manager.

P2P Protocol

P2P Protocol is implemented over HTTP/2 standards and uses Google RPC. It provides many capabilities like bi-direction streaming, flow control and multiplexing requests over a single connection. It works with existing Internet infrastructure, including firewalls, proxies and security. P2P component defines messages used by peer nodes, from point to point to multicast.

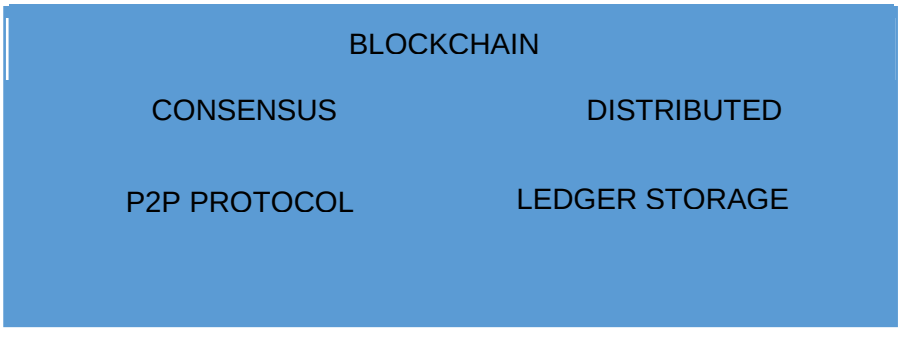
Distributed Ledger

Distributed Ledger manages the world state and blockchain. It implements three key attributes. It efficiently calculates the cryptographic hash of the entire dataset of each block. It efficiently transmits a minimal "delta" changes to the dataset, when a peer is out of sync and needs to "catch up". It minimizes the amount of stored data required for each peer to operate.

Distributed Ledger uses RocksDB to persist the dataset and to build an internal data structure.

Consensus Manager

Consensus Manager defines the interface between the consensus algorithm and the other Hyperledger components. The consensus manager receives transactions and depending on the algorithm decides how to organize and execute the transaction. Successful execution of transaction results in changes to the ledger.



Transaction

Hyperledger supports two types of transactions

Code deploying transaction

Code deploying transaction submits, updates or terminates a chaincode. The validating nodes protects the authenticity and integrity of the code and its executing environment.

Code invoking transaction

Code invoking transaction is an API call to a chaincode function. It is similar to how a URI invokes a servlet in JEE. Each chaincode maintains its own state and a function call is made to trigger chaincode state changes.

Event stream in a decentralized network is complex in nature. An event can appear to occur multiple times once on each peer node and callbacks can end up receiving multiple invocations for the same event. Therefore, a peer node manages the events that the applications are interacting with. The peer nodes state events as their conditions are satisfied, in no special order. If events do not continue to be created, the application can capture events if required.

Chaincode

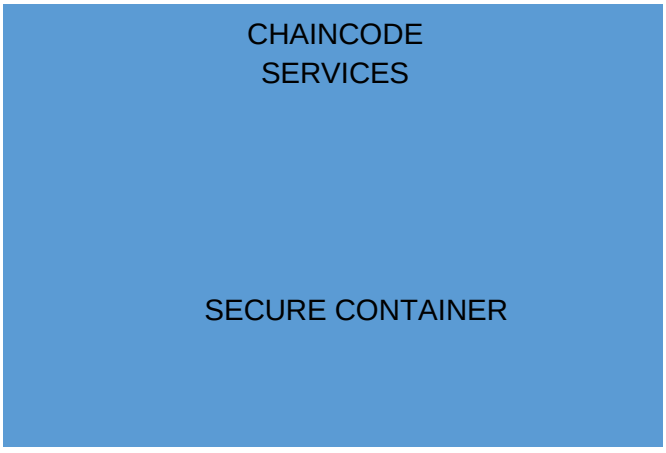
Chaincode is a decentralized transactional program, running on the validating nodes.

Secure Container

Chaincode Services uses Docker to host the chaincode without relying on any virtual machine or computer language. Docker provides a secured, lightweight method to sandbox chaincode execution. The environment is a "locked down" and secured container, along with a set of signed base images containing secure OS and chaincode language, runtime and SDK images for Golang. Additional programming language scan be enabled, if required.

Secure Registry

Secure Registry Services enables Secured Docker Registry of base Hyperledger images and custom images containing chaincodes.



Hyperledger includes the REST and JSON RPC APIs, events, and an SDK for applications to communicate with the network. Typically, applications interact with a peer node, which requires some form of authentication to ensure that the entity has proper privilege; messages from a client are signed by the client identity and verified by the peer node. Hyperledger provides a set of CLIs to administer and manage the network. CLI can also be used during development to test chaincodes. REST API and SDK are built on top of JSONRPC API, which is the most complete API layer. SDK will be available in Go (Golang), JavaScript, and Java; additional programming languages can be added as necessary.

Hyperledger Fabric

Hyperledger Fabric was developed by IBM, and is used to develop blockchain applications with a modular architecture, so that components such as consensus and membership services can be plug-and-play. A fabric model consists of peer of nodes which executes smart contracts, accesses ledger data, endorse transaction and interface with applications.

The fabric framework is implemented on Go. It enables consortium blockchains with different level of permissions. Fabric uses container to host smart contracts that has the business logics of the application.

Lab 1: Building a Use Case

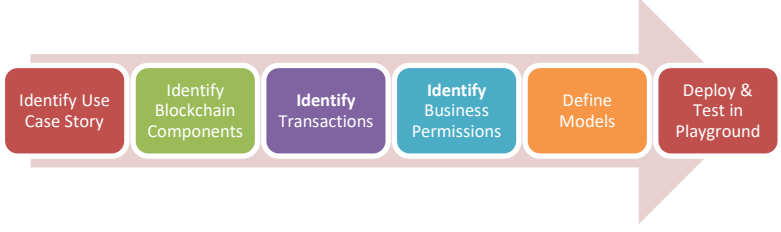
Objectives

After you completed all the following chapters with lab exercises, you will be able to:

- Have Development Environment Setup for Hyperledger Fabric
- Understand on types of nodes & their functions
- Create permissioned blockchain network
- Use Authorizations and authentication
- Write chaincode and call transactions via composer tools and rest server
- Understand different Hyperledger architecture templates

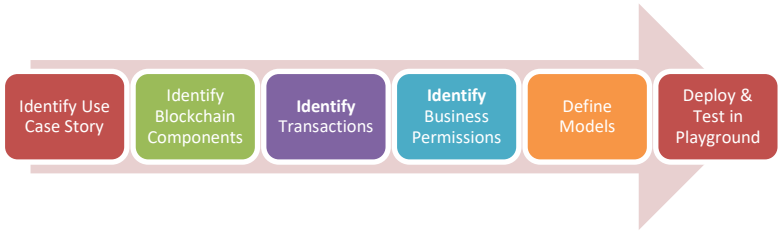
Tasks

1. Identify Use Case Story
2. Identify Blockchain Components
3. Identify Transactions
4. Identify Business Permissions
5. Define Models
6. Deploy & Test in Playground

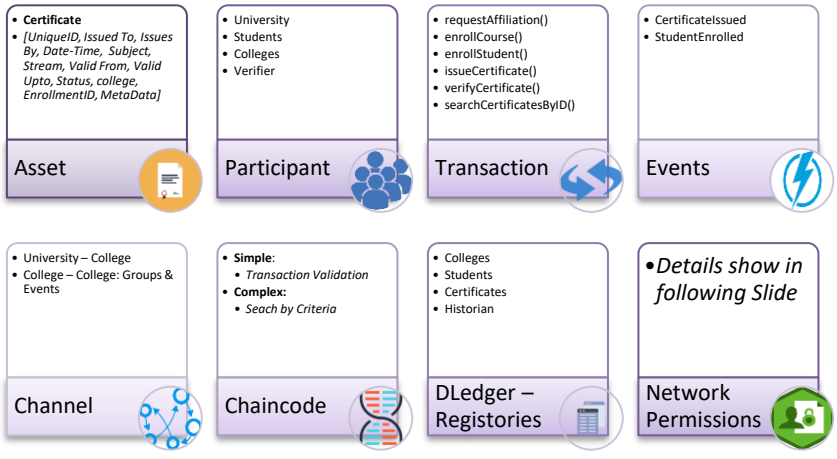


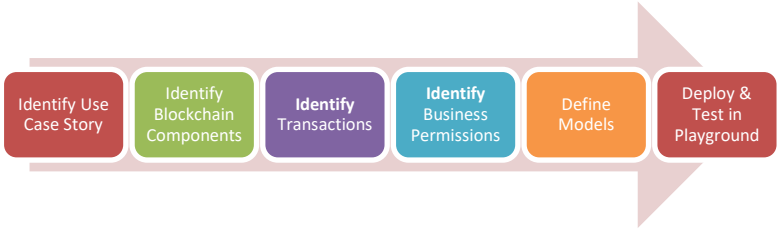
TASK#1: Private Blockchain Use Case & Stories

- **University**: As a University, I am an institution for higher studies. I want to affiliate Colleges and courses. Enroll Students and Award certificates
- **Students**: As a student I want to take admission, enroll for a specific education program and get awarded with certification upon successful completion
- **Colleges**: As a college, I provide students specialized courses as per the guidelines/affiliation with university.
- **Verifier**: As a verifier, I want to verify if the certificate ID (public address) is valid and its corresponding details (Issued to, Status, etc). I can also search all certificates associated with a Nation-ID (or tokenized ID)
- **Business Network Admin (BNA)**: As a network admin I want to deploy code and runtimes.



TASK#2: Identify Blockchain Components





TASK#3: Identify Transactions

Step 1:

requestAffiliation()
Input:

- College Detail
Output: *Status*
By Participant: *college*
On Participant: *University*

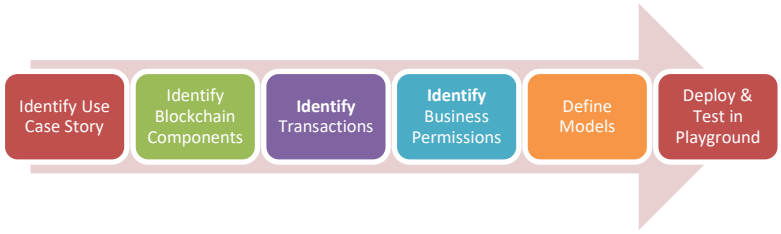
searchCertificateByID()
Input:
- National-ID / Enrollment-ID
Output: Certificate *List*
By Participant: *Verifier*
On Participant: *Blockchain*

enrollCourse()
Input:
- Course Detail
Output: *CourseID*
By Participant: *College*
On Participant: *University*

issueCertificate()
Input:
- Certificate Detail
Output: *Public Address*
By Participant: *College*
On Participant: *University*

enrollStudent()
Input:
- Student Detail
Output: *Student EnrollmentID*
By Participant: *College*
On Participant: *University*

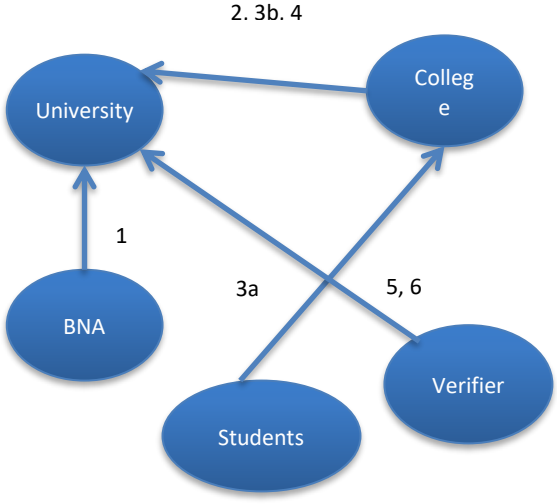
renewCertificate()
Input:
- Certificate Detail
Output: *Status-Success/fail*
By Participant: *College*
On Participant: *University*

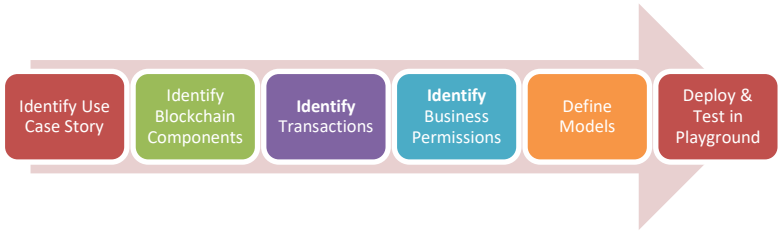


verifyCertificate()
Input: *Public Address*
Output: *Certifcate Details*
By Participant: *Verifier (anyone)*

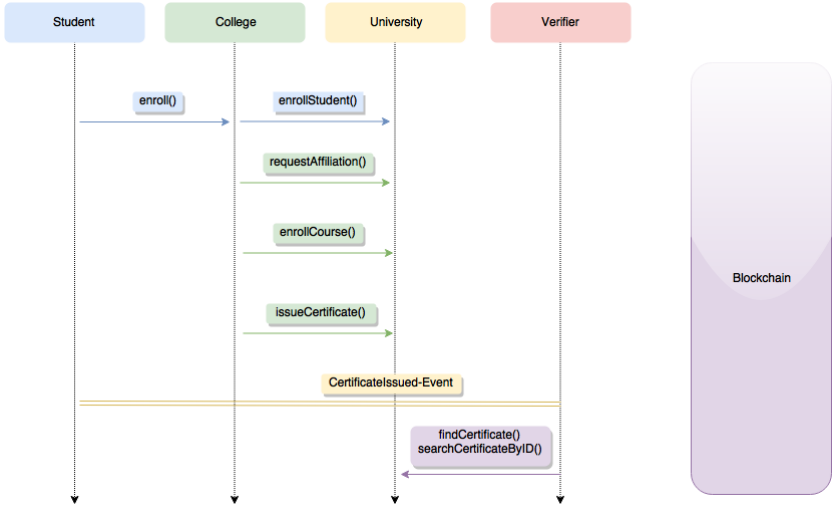
Step #3:

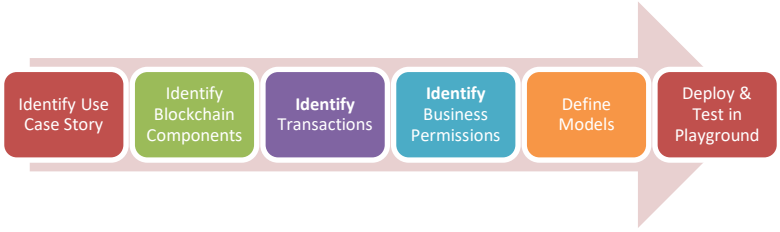
1. requestAffiliation()
2. enrollCourse()
3. enrollStudent()
4. issueCertificate()
5. verifyCertificate()
6. searchCertificatesByID()



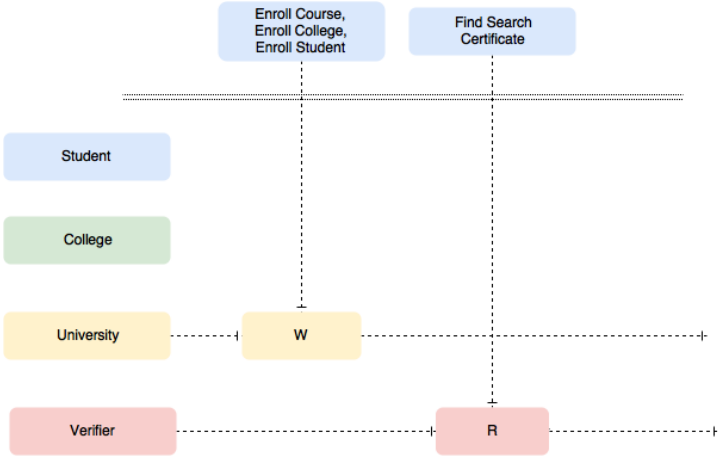

Identify Use Case Story
Identify Blockchain Components
Identify Transactions
Identify Business Permissions
Define Models
Deploy & Test in Playground

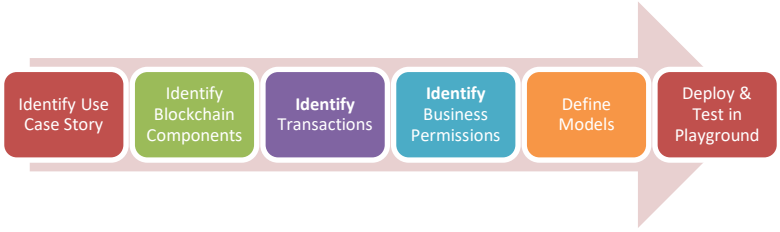
Step #4:


Student
College
University
Verifier
enroll()
enrollStudent()
requestAffiliation()
enrollCourse()
issueCertificate()
CertificateIssued-Event
findCertificate()
searchCertificateByID()
Blockchain



TASK#4: Identify Business Permissions





TASK#5: Define Models

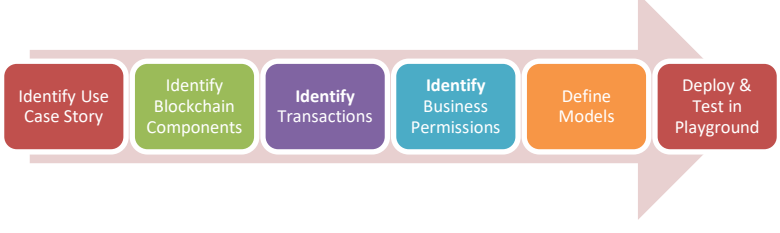
Step 1: Asset

- The main asset that we are dealing with is the Certificate that we will store in the blockchain and will be encrypted & digitally signed

```
/**
 * Digital Certificate Asset
 */
asset Certificate identified by certificateId {
  o String certificateId
  --> College college
  --> Student issuedTo
  --> Program program
  o String issuedBy
  o DateTime issuedDate
  o DateTime validUpto
  o String currentStatus
  o String metaData
}
```

- Note: (→) Defines the reference to other previously defined Network classes

```
/**
 * An abstract for all other intermediate assets
 */
abstract asset assetBase {
    o String memberId
    o String name
}

asset Program identified by memberId extends assetBase {
}
```



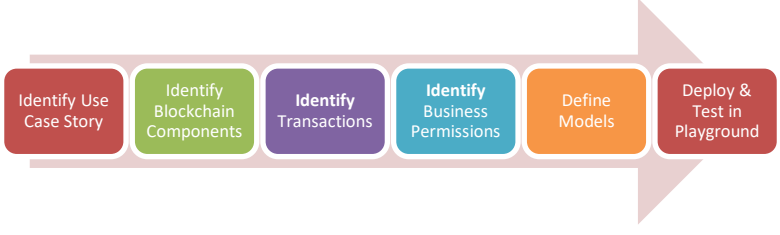
Step 2: Participants

- In the simple Use Case of university we have Members with following properties
    - Identified by memberID
    - Has a Name
- As these properties and common within all participants let's encapsulate it in an abstract with name MemberParticipant

```
abstract participant MemberParticipant {
    o String memberId
    o String name
}
```

- We then extend all particiapnts with this base MemberParticipant abstract

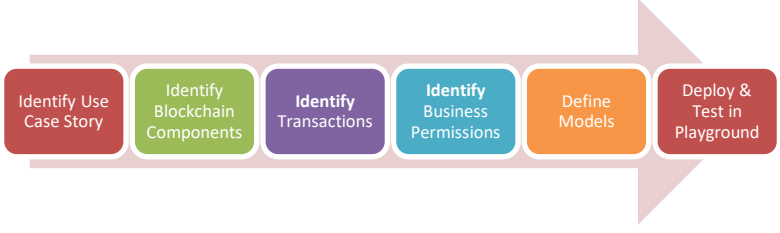
```
participant University identified by memberId extends MemberParticipant {
}

participant Student identified by memberId extends MemberParticipant {
    o String surname
    o DateTime dob
}

participant College identified by memberId extends MemberParticipant {
}

participant Verifier identified by memberId extends MemberParticipant {
}
```



Step 3: Transactions

- Transactions can be defined in a similar modeling language
- We need to define all the parameters that are needed to complete the requested transaction

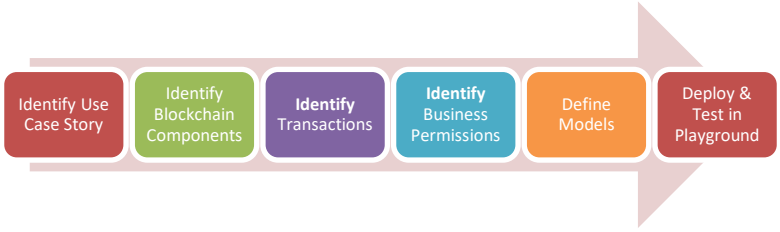
```
transaction issueCertificate {
    o Asset Certificate
}

transaction requestAffiliation {
    o String name
}

transaction enrollProgram {
    o String name
}

transaction enrollStudent {
    o String name
}

transaction verifyCertificate {
    o String certificateId
}

transaction searchCertificatesByID {
    o String enrollmentId
}
```



Step 4: Events

- Events are also declared similar to transactions and defines parameters that is broadcasted with the event.

```
event CerticateIssued {
    o String certificateId
}

event StudentEnrolled {
    o String studentID
}
```



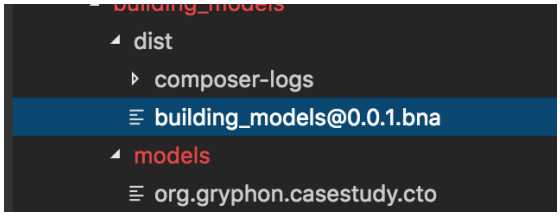
TASK#6: Deploy & Testing in Playground

Step 1: Create Archive File

- Copy and unzip chapter1.zip to a folder
- CD into the 'chapter1' folder
- Create a folder 'dist' and cd into it
- Now type the following command to create an archive file

```
> composer archive create –t dir –n ../
```

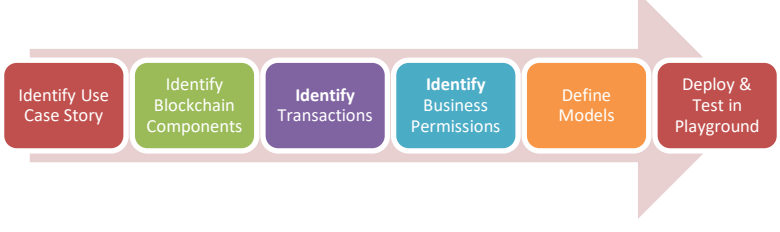
- This will create a .bna file in the dist folder which contain all the created model files in its binary (.cto : Business model files)



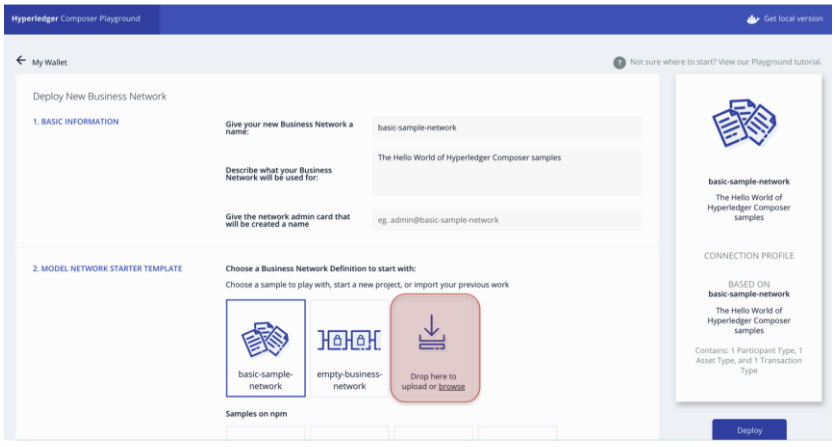
Step 2: Open your Internet browser and navigate to the following website:

https://composer-playground.mybluemix.net/

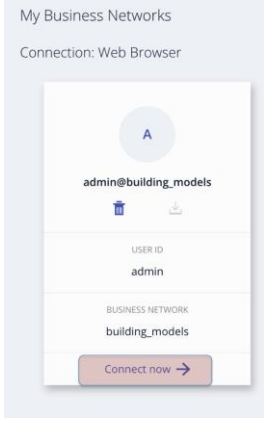
It will take a while to prepare your browser

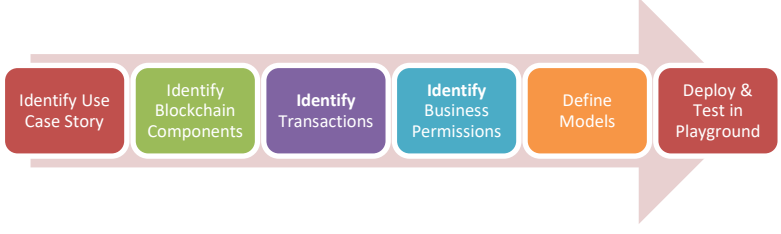


Use the deploy button as shown in figure below to deploy the archive



After deploying, start the application click the highlighted button as in figure below;

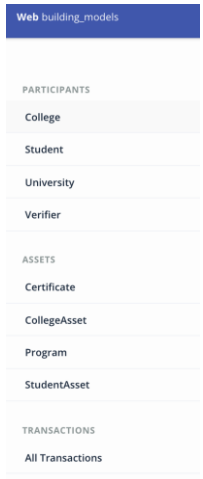




Step 3: Test using playground

Browse to the test panel as in figure below;

Left column will show the entire asset, participants, which you can create and Test



SUMMARY

- Hyperledger is a Distributed Ledger Technology for the business
- Chaincode automates the Business processes
- Four Essential characteristics that make HyperLedger Suitable for Business are
    - Permissioned Network
    - Transaction confidentiality
    - No Mining needed compared to peers
    - Programmable using chaincode

REFERENCES

- http://hyperledger.org
- http://hyperledger.org/projects/fabric
- https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html

CHAPTER 4: HYPERLEDGER FABRIC FUNDAMENTALS

Theory

In this chapter, we will cover the fundamental blocks of Hyperledger Fabric, discuss about what constitutes of the distributed ledger in Fabric, types of node in Fabric, endorsement policies, membership cards and the certificate authority.

Fabric Definitions

Transactions

A transaction is invoked by the SDK and sent to the endorser to verify it and authenticate it. Transaction either creates new chaincode or invoke transactions on already deployed chaincode.

Ledger

Ledger provides the provenance capability in blockchain. All the peers in a network contain the ledger. An orderer too can have ledgers. In the reference of peer, we refer the ledger as Peer Ledger and in the reference of orderer it is referred as orderer Ledger.

Ledger is constructed by the ordering services as ordered hash chain of blocks of valid and invalid transactions. The hash chain enforces the total order of blocks in a ledger and each of these blocks contain an array of ordered transaction.

Nodes

Nodes in a network are the communication entities in a blockchain. There are three kinds of nodes:

Client

Client represents the entity that acts on behalf of the end user. It submits the actual transaction-invocation to the endorser and broadcasts all the transaction proposals to the ordering services. The client may connect to any of the peers of its choice.

Peer

A peer receives updates in the form of blocks from the ordering services and maintain the ledger. Peers can also take up duties of the endorser.

Endorser

Endorsing peer is the gateway to the start with the Hyperledger Fabric. It authorizes a given transaction before it is committed. Every chaincode must specify an endorsement that refer to a set of endorsing peers. This policy describes the conditions for the valid transaction endorsement.

Ordering nodes

Ordering services handles the main function of the Hyperledger Fabric to maintain consistency in the transactions processes. Ordering services provides a shared communication to clients and peers and helps in broadcasting the messages containing transactions.

The channels can be defined as per the requirement provided. It ensures privacy in a network. Ordering services may support many channels. Client can connect to any given channel and can send and obtain messages.

Ordering Services API

Ordering service provides peers with an interface that helps connect to the channel provided by the ordering service.

Broadcast

A client calls this to broadcast an arbitrary message blob for dissemination over the channel. This is also called request(blob) in the BFT context, when sending a request to a service.

Deliver

The ordering service calls this on the peer to deliver the message blob with the specified non-negative integer sequence number and hash of the most recently delivered blob. In other words, it is an output event from the ordering service. deliver() is also sometimes called notify() in pub-sub systems or commit() in BFT systems.

Channels

Channels provide the way of communication of private and confidential transaction across a network. The channel is defined by each component in the network anchor peers per member, shared ledger, chaincode application and the ordering service nodes. Each party must be authorized to transact on the channel.

Certificate Authority

Certificate Authority provides certificate services to blockchain users. These services provide services like user enrollment, transaction invoked, and TLS se
cured connection between users or components of blockchain.

In the lab section we will create our first business network, model our first university use-case, write chaincode and its test.

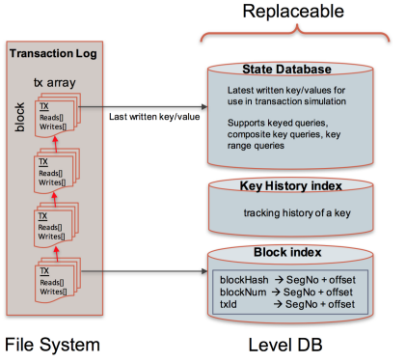
Finally, we shall deploy the created business network and test it using chai and mocha framework. Also use explorer to add entries to the asset and validate the transaction history using the Historian registry and explorer.

Distributed Ledger

The ledger is the sequenced, tamper-resistant record of all state transitions. State transitions are a result of chaincode invocations (“transactions”) submitted by participating parties. Each transaction results in a set of asset key-value pairs that are committed to the ledger as creates, updates, or deletes.

In Fabric Ledger has two parts:

- **State data**: Representation of current state of the assets
- **Transaction Logs**: Record of all the transactions which modified the state data



	Transaction Logs:	**State data**
Type	Is Immutable	Mutable
Operations	Create, Retrive	ALL – CRUD
DB	levelDB	levelDB / CouchDB
Behaviour	Embedded within Peers	Key-value paired (Json or Binary)
Query	Supports Simple Query	CouchDB Supports complex queries

Each transaction has a unique ID, its time-stamped and contains signatures of every endorsing peer and are submitted to ordering service

The ledger is comprised of a blockchain (‘chain’) to store the immutable, sequenced record in blocks, as well as a state database to maintain current Fabric state. There is one ledger per channel. Each peer maintains a copy of the ledger for each channel of which they are member.

Nodes:

The concept of node is common in all blockchain technologies. Node becomes the communication end point in blockchain technology. Nodes connect to other nodes and that is how a blockchain is formed.

Nodes use a type of peer-to-peer protocol for keeping the distributed ledger in sync across the network.

In public blockchain like Ethereum; anyone can participate as a node by downloading node client called wallet. But in the case of a permissioned Hyperledger network, things are quite different.

In Hyperledger, nodes need valid certificate to be able to communicate to the network and the participants use applications that connect to network by way of the nodes. Participant's identity is not the same as the nodes identity. The participant that executes or invokes a transaction their certificate is used for signing that transaction. The network to check if they should trust the node uses node's certificate or not. In case the nodes certificate is revoked or has expired in that case the transaction thought signed by a valid certificate held by the participant is broadcasted to the network, but the transaction will be rejected because the certificate that node is using has been expired or has been revoked.

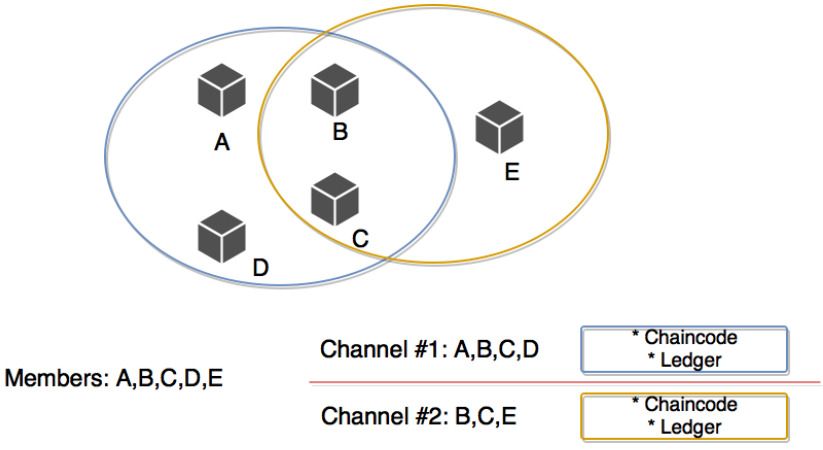
In Hyperledger, all Nodes are NOT equal. There are three distinct types of nodes:

1. **Client Node:** That initiates the transaction
2. **Peer Nodes:** Commits Transaction & keeps the data in sync across the ledger
3. **Ordered:** They are the communication backbones and responsible for the distribution of the transactions

Channel:

Members can participate on multiple hyperledger blockchain networks. Transaction in each network is isolated and this is made possible by way of what is referred to the channels.

Channel is a data partitioning mechanism to control transaction visibility only to stakeholders. Other members on the network are not allowed to access the channel and will not see transactions on the channel.



A chaincode may be deployed on multiple channels, each instance is isolated within its channel. Similarly, each channel maintains their own ledger. (Hyperledger version 1.0)

Separation of the ledger, by defining the specific channel for each ledger and peer node

memberships are defined in the Chaincode configuration. It is stored in the Genesis block of the ledger, which also stores the members, policies, and anchor peers. The Genesis block defines the read/write access on a channel.

Peers are connected to the channel and can receive all the transactions that are broadcasted on that channel. Consensus takes place within a channel by members of the channel.

Node Types: Ordered, Anchor & Endorser

Client Node:

The client represents the entity that acts on behalf of an end-user. It must connect to a peer for communicating with the blockchain. The client may connect to any peer of its choice. Clients create and thereby invoke transactions.

Peer Node:

They are nodes that maintain the state and copy of a shared ledger. Peers are authenticated by certificates issued by MSP. In HyperLedger Fabric, there are three types of peer nodes depending upon the assigned roles:

Endorsing Peers (Endorsers)

An endorser executes and endorses transactions.

The endorsing peers take the role of endorsing transactions before they are ordered and committed as per the policy defined in Chaincode. The client application creating the transaction sends it to the endorsing peers as per the policy in chaincode. The endorsement policy is instantiated at the chaincode of the client application and forwarded to the endorsing peers. The endorsing peer evaluates and validates the transaction and produces an endorsement signature and then returns it to the application. There may be one or more pre-specified set of endorsing peers involved as per the endorsement policy. The transaction is evaluated and declared valid only if it has been endorsed by the endorsing peers as per policy.

Ordering Service Nodes (Orderers)

- Responsible for consistent ledger state across the network
    - Consensus Mechanism
    - Ensures order of Transactions
- Creates Blocks & Provides atomic delivery/broadcast
- Message Oriented Middleware options for orderer service in Hyperledger:
    - SOLO: Single Node (Good for Development)
    - Kafka: High throughput, scalable & Fault Tolerant

All transactions from the network are received by the orderer, which orders and groups them, then packages the transactions and creates blocks. The orderer service delivers blocks to the committing peers that are allowed to be part of a Channel. The orderer services do not review transaction information. The orderer makes guaranteed atomic delivery of blocks to the committing peers on the channel.

The orderer supports multiple channels using a publish/subscribe messaging system (based on Apache Kafka and Zookeeper). The ordered provides a practical Byzantine Fault tolerance for failures without a single-point of failure.

Ordering service nodes also provide the following services:

- Authentication of clients
- Maintenance of a system chain that defines ordering service configurations, root certs and MSP IDs for authenticated organizations and a grouping of profiles containing the various consortia within the network.
- Filtering and validation for configuration transactions that reconfigure or create a channel.

Committing Peers (Committers)

Verifies endorsements and validates transaction results

The committing peers receive blocks from the Orderer service, which have already been endorsed by the endorsing peers. The Committing peers ultimately commit the transactional state by adding the blocks to the ledger. Before committal, the peers validate or invalidate the transaction by verifying if the endorsement policies are met, authenticate the signatures, and also verify the version info (if there is any double spending).

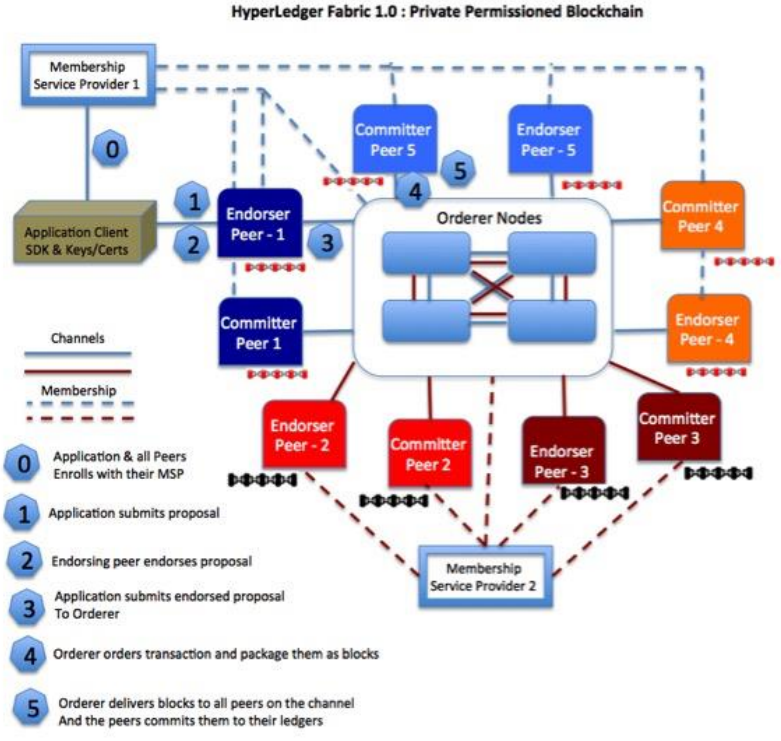
HyperLedger Transaction Flow:

Assuming the HyperLedger Fabric 1.0 up and running, in a typical transaction flow of asset exchange:

- All application users and peer node members are registered in the MSP and issued with Keys/certificates from the CA for authenticating the network. The Chaincode representing the initial state is installed on the peers and the channels

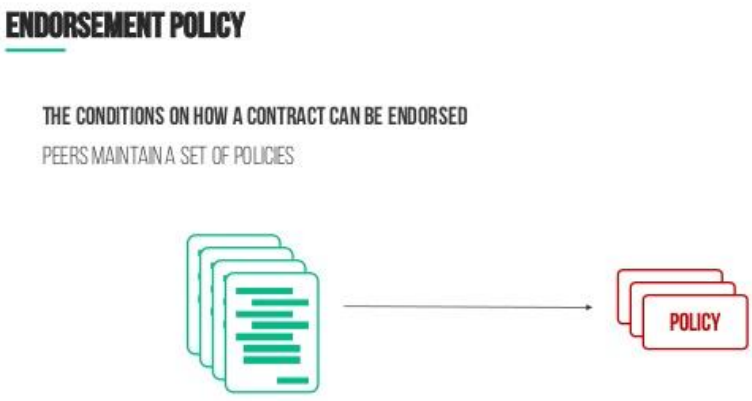
are active.

- The application client initiates a transaction (Client A makes a request to Client B to transfer an asset). The endorsement policy states that the request must be endorsed by Peer A and Peer B.
- The application submits a transaction proposal to the endorsing peers A and B.
- The endorsing peers receive and verify the transaction proposal and its signature, then executes the transaction and return a signed proposal response back to the application client.
- The application client verifies the responses from the endorsing peers. It assembles the response into a transaction and sends it to the Orderer service.
- The Orderer services order the transactions chronologically and package those transactions as blocks specific to a channel.
- The Orderer service delivers the blocks of the transactions to all the peers on the channel.
- The peers perform the validation of the blocks for endorsement policy, signatures, verification and version info, and finally appends the block to the chain and commits the state to the database. It then notifies the application client.



Endorsement Policies

An endorsement policy is a condition which *endorses* a transaction. Blockchain peers have a pre-specified set of endorsement policies, which are referenced by a deploy transaction that installs specific chaincode. Endorsement policies can be parameterized, and these parameters can be specified by a deploy transaction.



To guarantee blockchain and security properties, the set of endorsement policies should be a set of proven policies with a limited set of functions to ensure bounded execution time (termination), determinism, performance and security guarantees.

Dynamic addition of endorsement policies (e.g., by deploy transaction on chaincode deploy time) is very sensitive in terms of bounded policy evaluation time (termination), determinism, performance and security guarantees. Therefore, dynamic addition of endorsement policies is not allowed, but may be supported in the future.

Transaction evaluation against endorsement policy

A transaction is declared valid only if it has been endorsed according to the policy. An invoked transaction on a chaincode must first obtain an *endorsement* that satisfies the chaincodes policy or it will not be committed. This takes place through the interaction between the submitting client and the endorsing peers as explained in Section 2.

Formally, the endorsement policy is predicated on the endorsement, and potentially further state that evaluates to TRUE or FALSE. For deployed transactions the endorsement is obtained according to a system-wide policy (for example, from the system chaincode).

An endorsement policy predicate refers to certain variables. Potentially it may refer to:

1. Keys or identities relating to the chaincode (found in the metadata of the chaincode), for example, a set of endorsers;
2. Further metadata of the chaincode;
3. Elements of the endorsement and endorsement, tran-proposal;
4. And potentially more.

The above list is ordered by increasing expressiveness and complexity, that is, it will be relatively simple to support policies that only refer to keys and identities of nodes.

The evaluation of an endorsement policy predicate must be deterministic. An endorsement shall be evaluated locally by every peer, such that a peer does *not* need to

interact with other peers, yet all correct peers evaluate the endorsement policy in the same way.

Membership Service Provider (MSP)

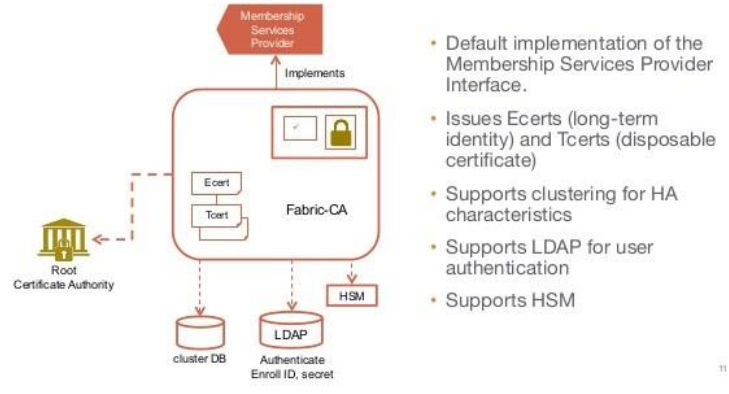
“Abstract component of the system that provides credentials to the clients, and the peers for then to participate in the Hyperledger Fabric network”

MSP implementation is based on the PKI (Public Key Infrastructure).

Service it Provides:

- Authorization Service
  - Role based
  - Examples:
    - Can this user issue further identity?
    - Can user deploy chaincode?
- Authentication Service
  - Where users Identity gets validated
  - Examples:
    - Is the user’s/peer’s certificate valid?
    - Is peer allowed to participate?

Fabric-CA



Certificate Authority

As a platform for **permissioned** blockchain networks, Hyperledger Fabric includes a modular **Certificate Authority (CA)** component for managing the network identities of all member organizations and their users. The requirement for a permissioned identity for every user enables ACL-based control over network activity and guarantees that every transaction is ultimately traceable to a registered user.

- The CA (Fabric CA by default) issues a root certificate (rootCert) to each member (organization or individual) that is authorized to join the network.
- The CA also issues an enrollment certificate (eCert) to each member component, server side applications and occasionally end users.
- Each enrolled user is also granted an allocation of transaction certificates (tCerts). Each tCert authorizes one network transaction.

This certificate-based control over network membership and actions enables members to restrict access to private and confidential channels, applications, and data, by specific user identities.

Lab 2: CHAINCODE BASICS

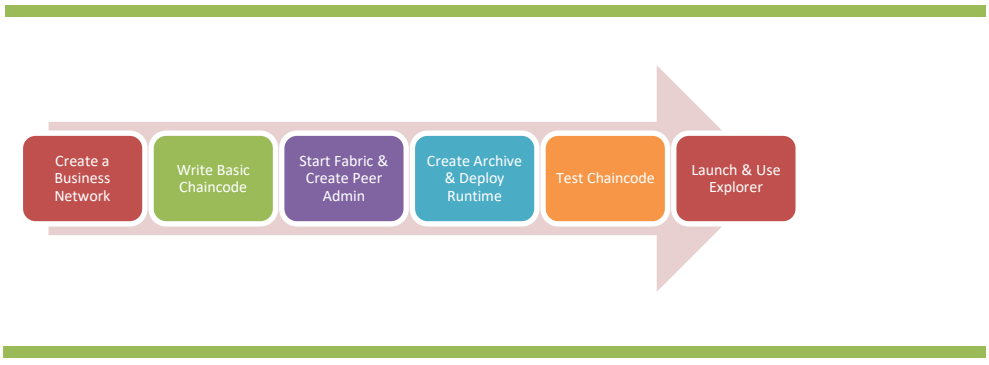
Objectives

After you have completed this lab exercise, you will be able to:

- Have a good understanding of Registries, distributed ledgers, types of nodes & their functions
- Create your first Hyperledger Business Network
- Start Hyperledger Fabric, create archive, deploy runtimes and execute the tests
- Use explorer to add / update asset, run transactions and validate general system business methods like historian and identities

Tasks

1. Create a Business Network using Yo Scaffolding
2. Write Basic Model + Chaincode + Test
    a. npm install
3. Start Fabric and Create PeerAdmin Card
4. Deploy Runtime
    a. Create Archive
    b. Import Admin Card
    c. Deploy To Runtime
5. Testing Chaincode
6. Running Explorer
    a. Launch explorer
    b. Use Explorer to add a asset
    c. Run Transactions
    d. View Historian



TASK#1: Create a Business Network Scaffolding

Step 1: Create a new folder and cd into it. Launch terminal window and use following commands

```
mkdir chapter2
```

```
cd chapter2
```

Step 2: Use Yo-Generator to create a business Scaffolding. Using the terminal window enter following commands

```
yo hyperledger-composer
```

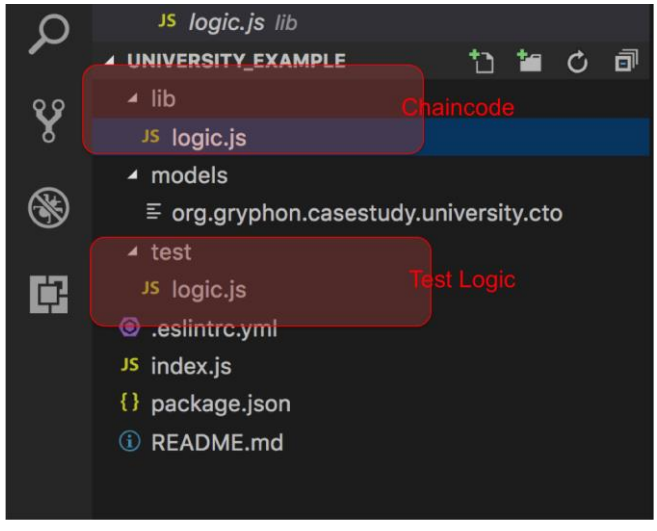
Step 3: Choose **'Business Network'** from the choice using your keyboard

```
Welcome to the Hyperledger Composer project generator
? Please select the type of project: (Use arrow keys)
  Angular
❯ Business Network
  Model
```

Step 4: After you have selected Business Network option you will be asked few details which you need to provide, shown below;


```
Welcome to the Hyperledger Composer project generator
? Please select the type of project: Business Network
You can run this generator using: 'yo hyperledger-composer:businessnetwork'
Welcome to the business network generator
[? Business network name: university_example
[? Description: Creating models for our university example
[? Author name:  Ernesto
[? Author email: university@example.com
[? License: GPL 1.0
[? Namespace: org.gryphon.casestudy.university
```

Step 5: Open the 'chapter2' folder in Visual Studio code to find the following directory structure



Task 1 is complete!

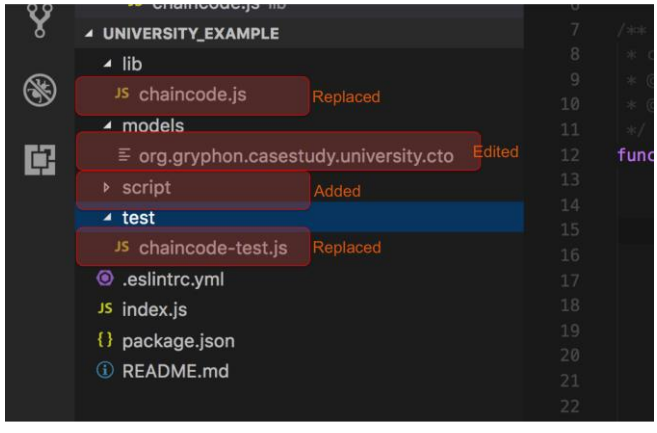
TASK#2: Write basic model, chaincode and test

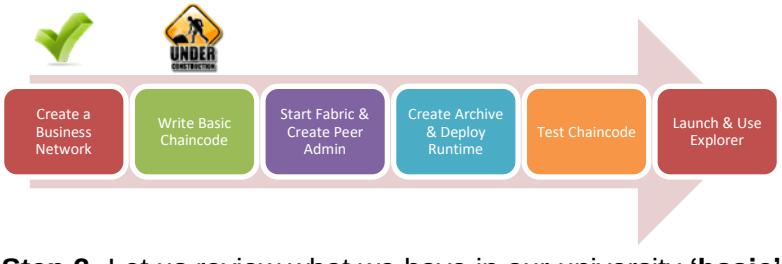
**** We will use the code 'chapter2' provided with this lab as a sample chaincode**

Step 1: Open **'Chapter 2'** code folder supplied with this tutorial using Visual Studio Code;

This folder is similar to the Business Network Scaffolding created earlier but with few changes as below;

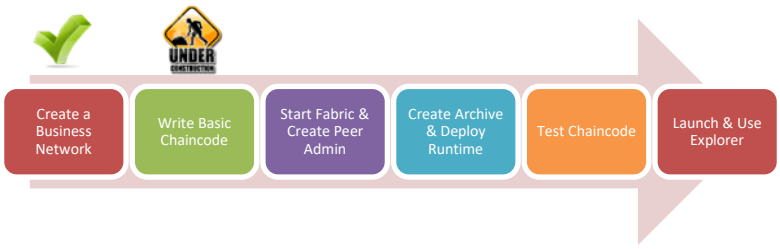
- 'logic.js' under lib folder is replaced with our university example chaincode and renamed to 'chaincode.js'
- 'logic.js' under test folder is replaced with 'chaincode-test.js' file which will test our university sample chaincode
- 'org.gryphon.casestudy.university.cto' under the models folder is modified to use a single asset in our university example
- Additional 'script' folder added to speed up the development process



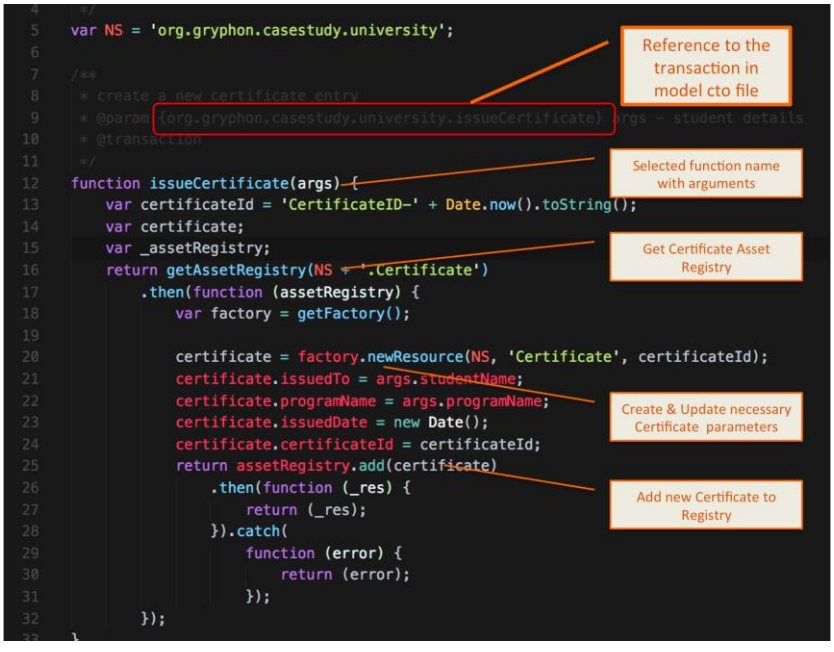


Step 2: Let us review what we have in our university **'basic'** example

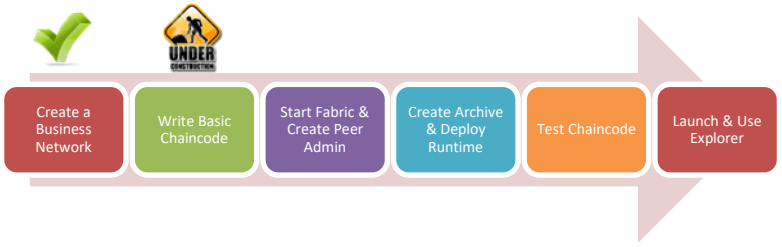
- Model file: org.gryphon.casestudy.university.cto

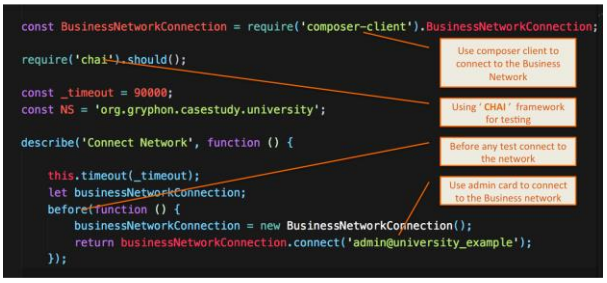


- Review Chaincode: ‘chaincode.js’



- Review Test code: ‘chaincode-test.js’





'chaincode-test.js' => continued…

Task 2 is complete!

TASK #3: Start Fabric and create peer admin card

Step 1: Launch terminal window and use following commands to change the directory to the installed fabric-tools folder

```
cd ~/fabric-tools/
```

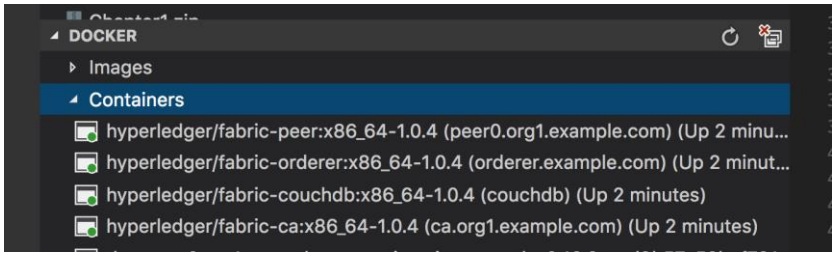
Step 2: Use the startFabric.sh script to start the fabric environment.

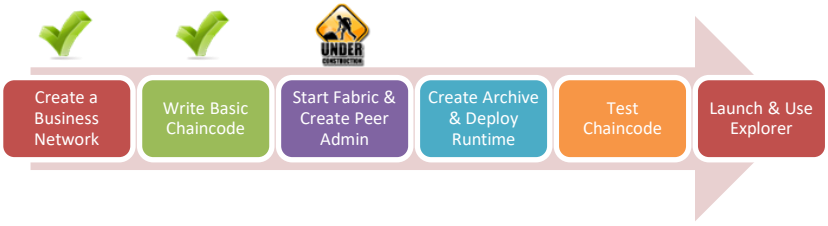
```
./startFabric.sh
```

This command will create and launch the necessary docker containers

```
ARCH=$ARCH docker-compose -f "${DIR}"/composer/docker-compose.yml up -d
Creating couchdb ... done
Creating peer0.org1.example.com ... done
Creating couchdb ...
Creating orderer.example.com ...
Creating peer0.org1.example.com ...
```

You can see containers in your Visual Studio Code (Docker Plugin installed)





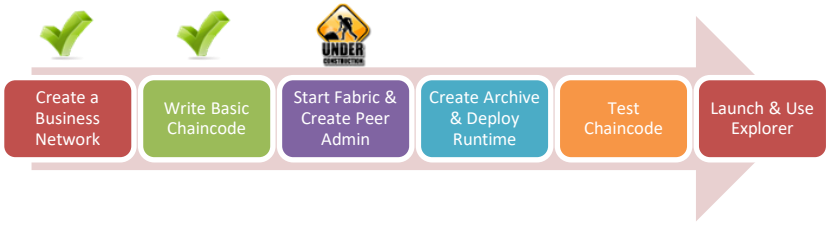
Step 3: Create Peer Admin Card to be able to deploy business runtimes.

From the fabric-tools directory use the createPeerAdmin.sh script

```
cd ~/fabric-tools/
```

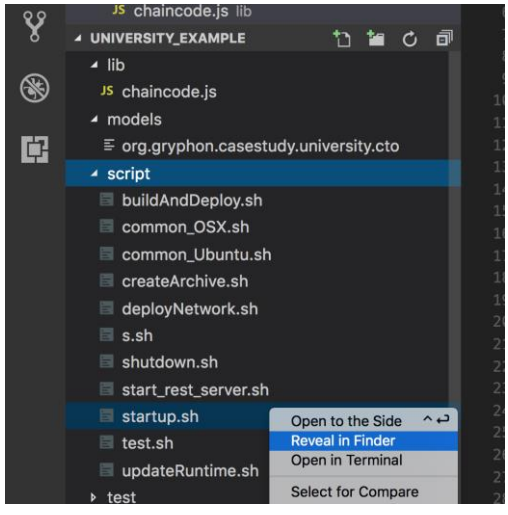
```
./createPeerAdminCard.sh
```

```
Successfully created business network card file to
        Output file: /tmp/PeerAdmin@hlfv1.card

Command succeeded


Successfully imported business network card
        Card file: /tmp/PeerAdmin@hlfv1.card
        Card name: PeerAdmin@hlfv1

Command succeeded

Hyperledger Composer PeerAdmin card has been imported
The following Business Network Cards are available:

Connection Profile: hlfv1

| PeerAdmin@hlfv1 | PeerAdmin |

Issue composer card list --name <Card Name> to get details a specific carc

Command succeeded
```



Step 4: To speed up the development process few scripts are available. You can use script in the 'Chapter 2' code folder to start Fabric and create peer admin card

***** From next time we will be using this script only to start Fabric and create peer card before deploying hence follow these steps *****



Change the path below in startup.sh script corresponding to your OS

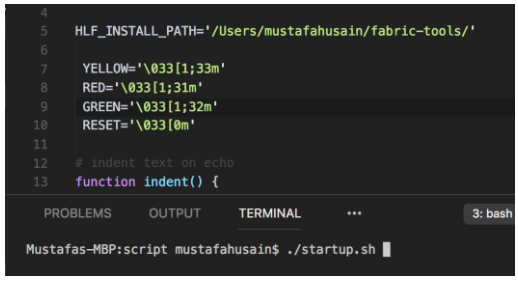
- **For MAC USERS**

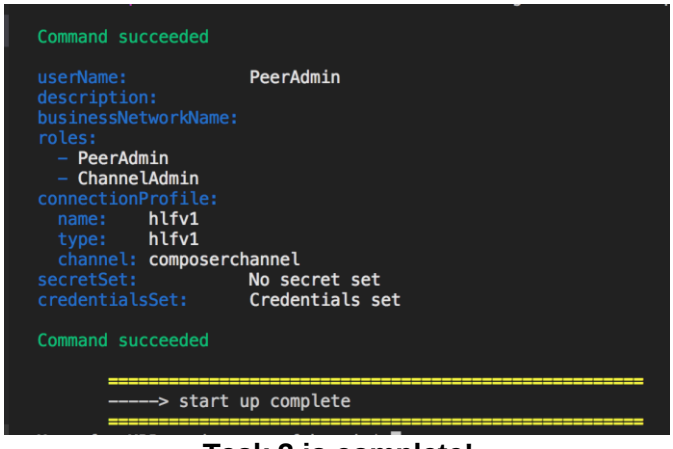
For Ubuntu Users:

```
HLF_INSTALL_PATH='~/(username)/fabric-tools/'
```

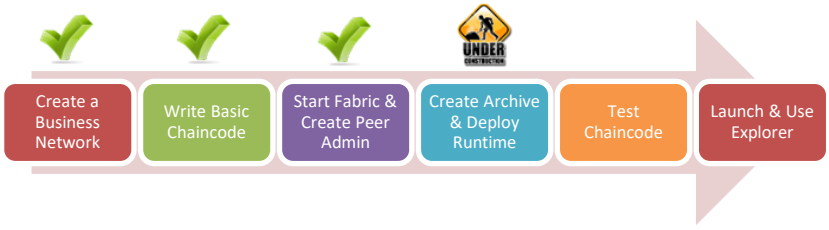
After this change has been done you can run this script from the visual studio terminal window directly

username-MBP:script username $ ./startup.sh





Task 3 is complete!



TASK #4: Create Archive & Deploy Runtime

Step 1: Open 'Chapter 2' code provided in Visual Studio Code. Open terminal window and cd into the 'Chapter 2' directory

Step 2: Create a 'dist' directory

```
mkdir dist
```

Step 3: Type the following command to create the archive

```
composer archive create -t dir -n . -a ./dist/university_example.bna
```

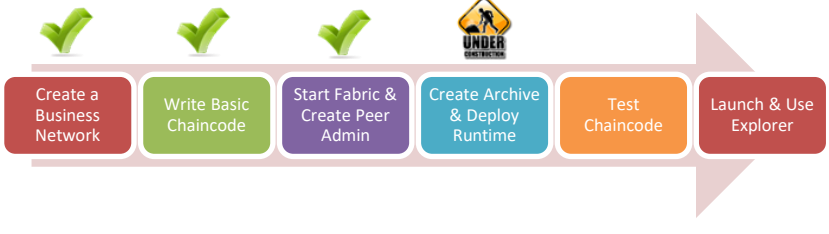
```
/university_example.bna
Creating Business Network Archive

Looking for package.json of Business Network Definition
        Input directory: /Users/mustafahusain/Desktop/university_example

Found:
        Description: Creating models for our university example
        Name: university_example
        Identifier: university_example@0.0.1

Written Business Network Definition Archive file to
        Output file: ./dist/university_example.bna

Command succeeded
```

This creates the archive **'university_example.bna'** in the ./dist folder created earlier.

Step 4: Now install the created peer admin card

First cd into the created 'dist' directory

```
cd dist
```



Type the following command in the terminal window

```
composer runtime install --card PeerAdmin@hlfv1 --businessNetworkName
university_example
```

This will take sometime and deploy the runtime

```
Mustafas-MBP:university_example mustafahusain$ cd dist/
Mustafas-MBP:dist mustafahusain$ composer runtime install --card PeerAdmin@hlfv1 --bu
etworkName university_example
✔ Installing runtime for business network university_example. This may take a minute.

Command succeeded
```

Step 5: Start Business network and create a network admin card to handle all network related operations

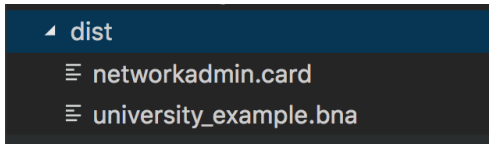
Type the following command in the terminal window and press enter:

```
composer network start -c PeerAdmin@hlfv1 -A admin -S adminpw -a
university_example.bna --file networkadmin.card
```

This uses the Peer Admin card to create the Network Admin cards

```
Starting business network from archive: university_example.bna
Business network definition:
        Identifier: university_example@0.0.1
        Description: Creating models for our university example

Processing these Network Admins:
        userName: admin

✔ Starting business network definition. This may take a minute...
Successfully created business network card:
        Filename: networkadmin.card

Command succeeded
```

This command will create a network admin card **'networkadmin.card'** in dist folder



Step 6: Import Business network admin card just created. Type the following command in terminal window

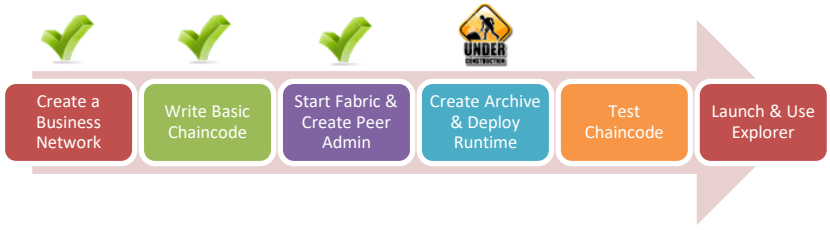
```
composer card import --file networkadmin.card
```

```
Successfully imported business network card
        Card file: networkadmin.card
        Card name: admin@university_example

Command succeeded
```

Step 7: To verify everything has been completed successfully we need to ping the network and check for success. Type the following command in terminal window

```
composer network ping --card admin@university_example
```

```
The connection to the network was successfully tested: university_example
        version: 0.16.3
        participant: org.hyperledger.composer.system.NetworkAdmin#admin

Command succeeded
```



[Alternative]: Doing all the steps one by one is cumbersome, hence to speed up the development process, a script has been written.

a. Move into the scripts folder of 'Chapter 2'

```
cd script
```

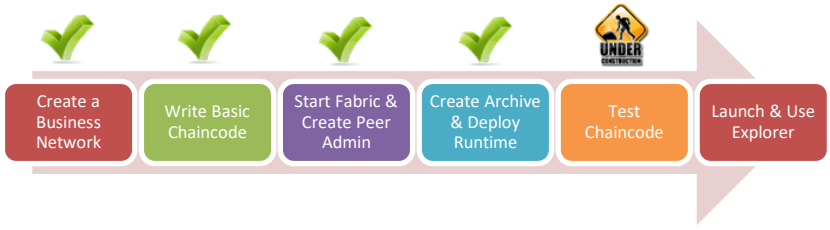
b. Run buildAndDeploy.sh script

```
./buildAndDeploy.sh
```

What does the script do?

- Everything done in Task #3 and #4 is done via this script
- Please ensure to have the correct path of hyperledger folder setup according to your operating system

Task 4 is complete!



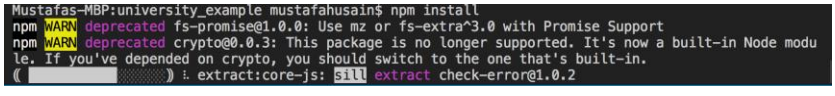
TASK#5: Running chaincode Test

Step 1: Open ‘Chapter 2’ code provided in Visual Studio Code. Open terminal window and cd into the 'Chapter 2’ directory

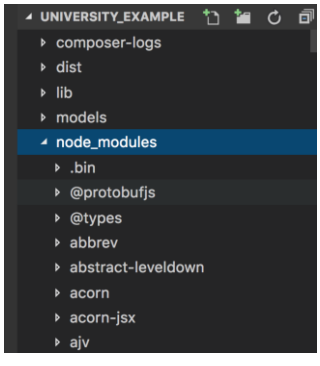
Step 2: Install the node modules by typing the following command into the terminal window

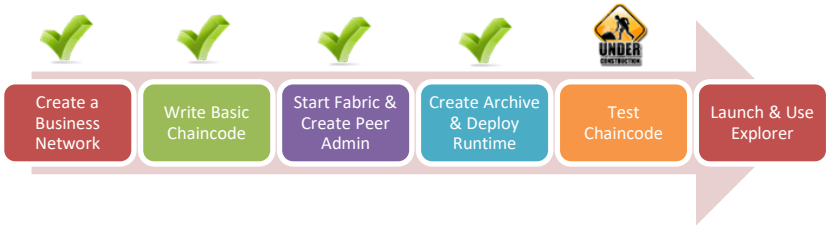
```
npm install
```

This will take a bit of time and install all the required node modules in the ./node_module folder



It may show some warnings but that should be OK.



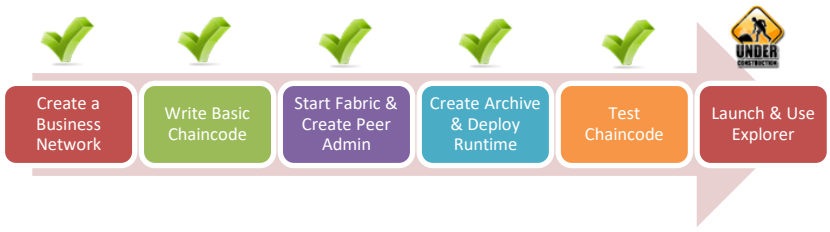


Step 3: After the node modules are installed type the following command to run the test

```
npm test
```

```
> university_example@0.0.1 test /Users/mustafahusain/Desktop/university_example
> mocha --recursive


  Connect Network
    #issueCertificate
Number of Certificates: 1
Certifcate ID: CertificateID-1521369285345
      ✓ should be able to issue Certificate (3101ms)


  1 passing (5s)
```

Task 5 is complete!



TASK#6: Launch & use explorer

Step 1: Open ‘Chapter 2’ code provided in Visual Studio Code. Open terminal window and cd into the 'Chapter 2’ directory

Step 2: Launch explorer using the following command

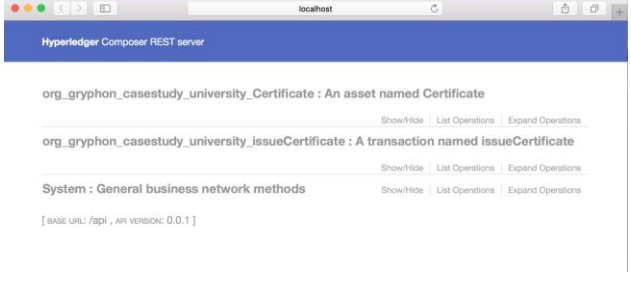
```
composer-rest-server -c "admin@university_example"
```

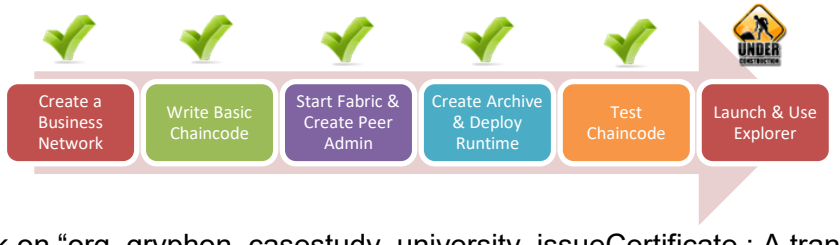
On successful completion of the command explorer rest server will be launched address as below;

```
Discovering types from business network definition ...
Discovered types from business network definition
Generating schemas for all types in business network definition ...
Generated schemas for all types in business network definition
Adding schemas for all types to Loopback ...
Added schemas for all types to Loopback
Web server listening at: http://localhost:3000
Browse your REST API at http://localhost:3000/explorer
```

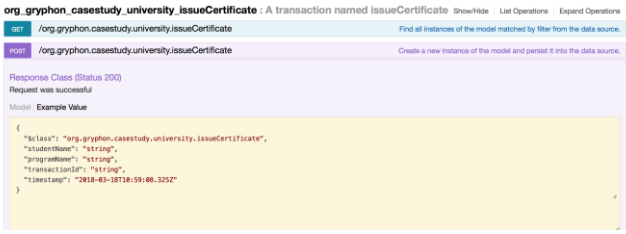
Step 3: Launch internet explorer and enter the following url:

http://localhost:3000/explorer/

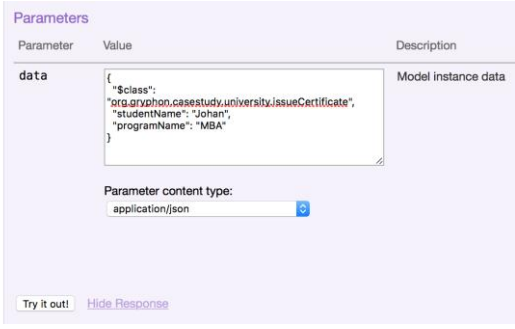




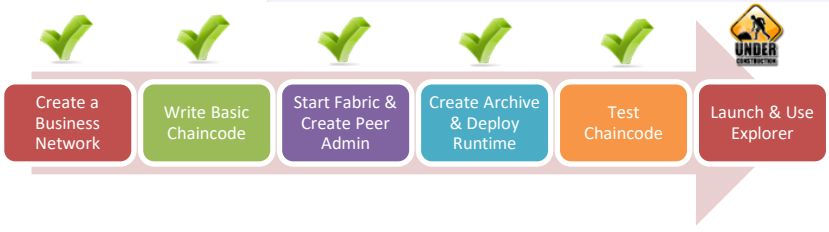
Step 4: Click on “org_gryphon_casestudy_university_issueCertificate : A transaction named issueCertificate”



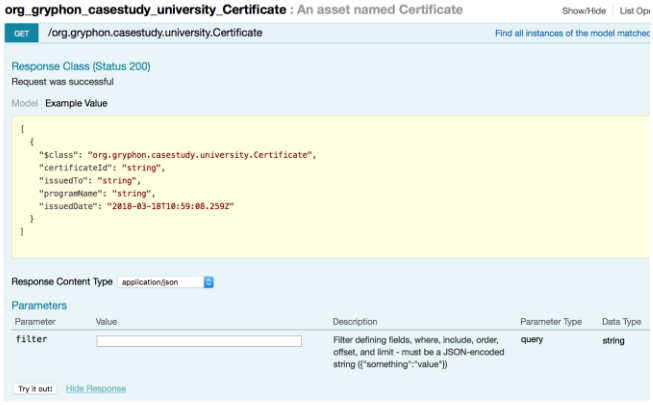
Using the example modify the transaction parameters and click “Try it out” as below



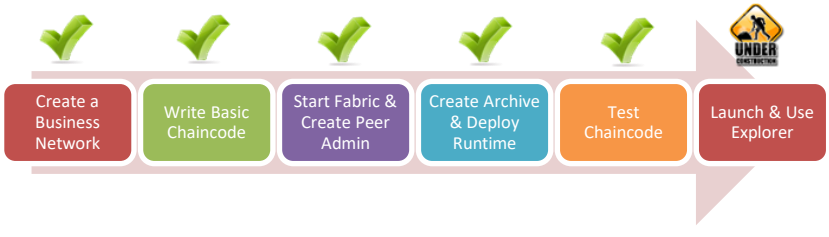
On clicking “Try it out” button you will response similar to below;



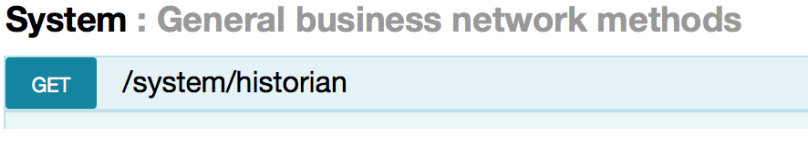
Step 5: Click on “org_gryphon_casestudy_university_Certificate : An asset named Certificate”



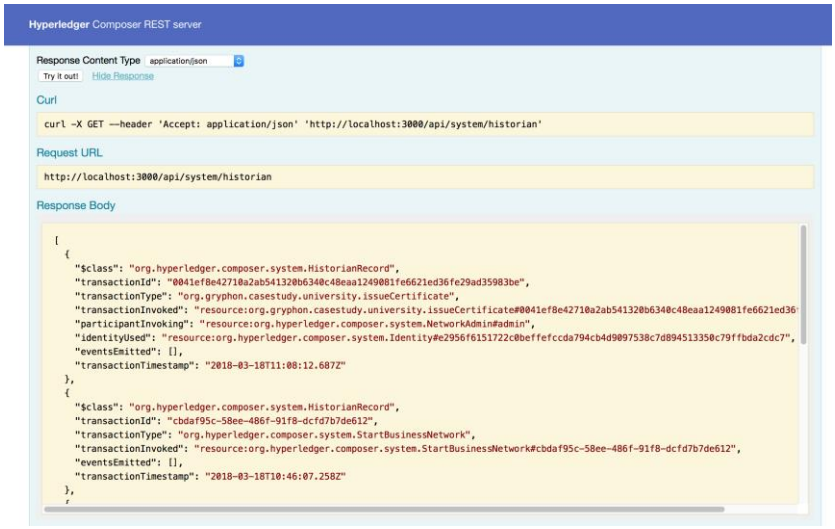
On clicking “Try it out” button you will receive the list of all the ‘Certificate’ Assets as below;



Step 6: Click on “System : General business network methods”. Click “Get” method and then click “Try it out!” button at the bottom;



Historian will list all the transactions done to date



Task 5 is complete!

SUMMARY

Hyperledger is a permissioned network. To be able to deploy chaincode and runtime network business cards are needed.

In this chapter we have learned the following:

- How to start Fabric composer
- Create Peer Admin Card and the Network Admin Card
- How chaincode is written and tested using chai framework
- Create archive and deploy the runtime
- How to use scripts for deploying and running during development process
- How use explorer to validate transactions, assets and the system methods

REFERENCES

- http://hyperledger.org
- http://hyperledger.org/projects/fabric
- https://hyperledger-fabric.readthedocs.io/

CHAPTER 5: PARTICIPANT, IDENTITIES & ACCESS CONTROL

Theory

In this chapter, we will cover access control and authorization, which are an important part of Hyperledger and the security architecture of a business network. Hyperledger enables an administrator to control the resources and/or data that a participant or participant role is authorized to see or do in a business network.

In the lab section we will explore and define an ACL file for our University example use-case and provide access permissions to different participants.

Access control rules (the language that defines ACLs) fall into two main areas:

- Authority to access system, network or administrative resources and operations in the System namespace (governing Network and System operations)
- Authority to access resources or perform operations within a given business network itself (like Create, Read, Update assets)

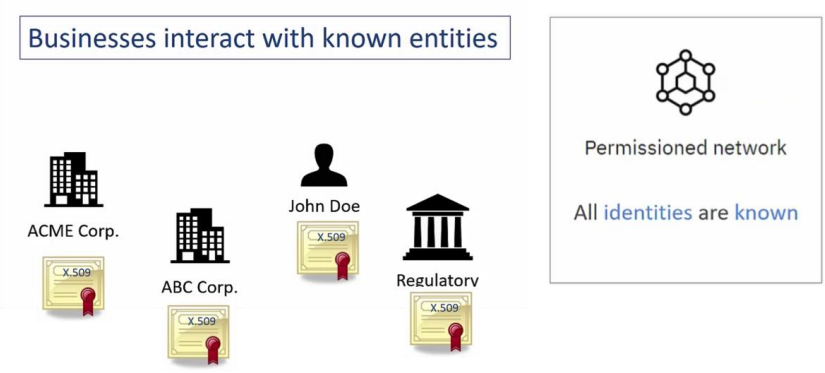
Finally, we will use the online Playground to try out some simple and conditional access rules. In doing so, we will interact with our University Use Case network as various identities. This will show how to apply access control to the users of the blockchain.

Participants and identities

A **Participant** is an *actor* in a business network. A participant might represent an individual or an organization. A participant can create assets and share assets with other participants. A participant can interact with assets by submitting transactions.

A participant has an identity set that can be validated to prove the identity of that participant.

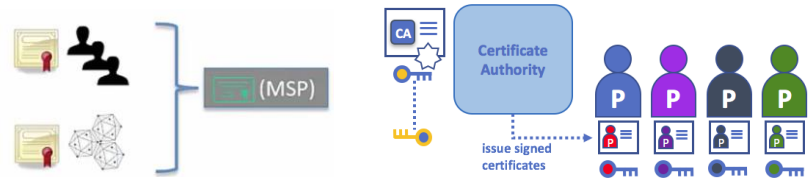
Hyperledger Fabric is a Private Permissioned Network for Businesses. Businesses interact with only known identities.



In Hyperledger, participants are separated from the set of identities that they can use to interact with a business network.

For a new participant to join a business network, a new instance of that participant must be created in the business network. The participant instance stores all of the required information about that participant, but it does not give that participant access to interact with the business network.

To grant the participant access to interact with the business network, an identity must be issued to that participant. The new participant can then use that identity document to interact with the business network.



An Identity usually expires after a specified period of time. An Identity may also be lost or stolen. If the identity expires, or if it needs to be replaced, then it must be Revoked so it can no longer be used to interact with the business network.

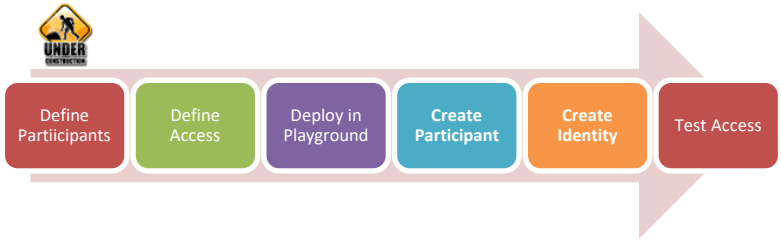
However, revoking an identity document does not remove the information about that participant and any assets that they own. Revoking the identity document simply removes the participant's ability to interact with the business network using that identity. Access to the business network can be restored by issuing the participant with a new identity.

These participant and identity management actions are performed by an existing participant in the business network, for example a regulatory body, or a participant in the same organization who has been trusted to manage participants/identities in that organization.

Lab 3: Adding Participants, Identities & Access Controls

Tasks

1. Define Participants and associated Transactions
2. Define Access (ACL File)
3. Deploy in Playground
4. Create Participant
5. Create Identity
6. Login To Business Network & Test Access



TASK#1: Define Participants & Transactions

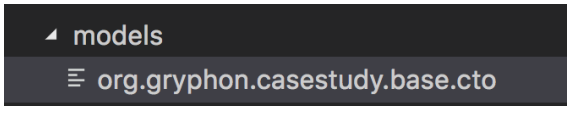
Step 1: Open 'Chapter 3' code provided in Visual Studio Code.

*#**Info**: 'Chapter 3' is a Business Network Scaffolding created as in Chapter#2-Task 1 with following changes:*

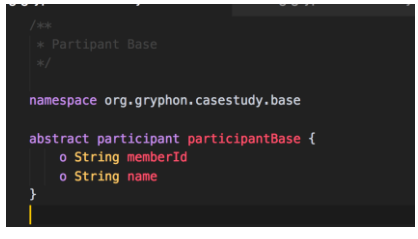
1. *Chaincode 'logic.js' is replaced with 'chaincode.js'*
2. *Test 'logic.js' is replaced with 'chaincode-test.js'*
3. *'script' folder is added*

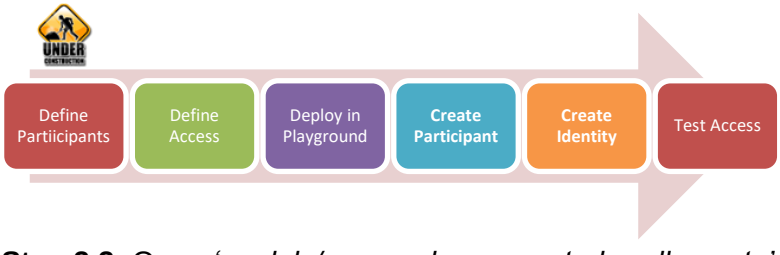
Step 2: Adding Participants:
We will segregate all the participants in different namespaces that helps in defining access.

Step 2.1: Open *'models/org.gryphon.casestudy.base.cto'* file in Visual Studio Code



Step 2.1: Defining abstract for class for all participants with mandatory common attributes

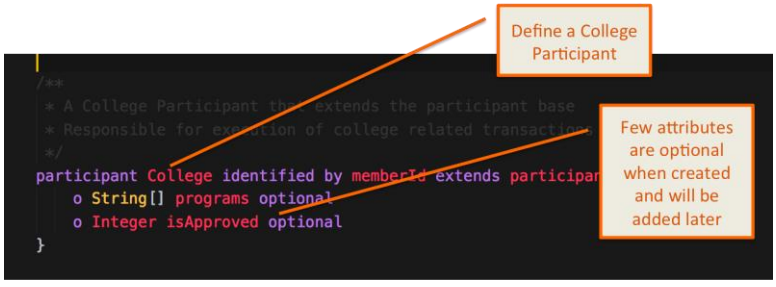




Step 2.2: Open *'models/org.gryphon.casestudy.college.cto'* file in Visual Studio Code. Define Namespace for college related participant and transactions.

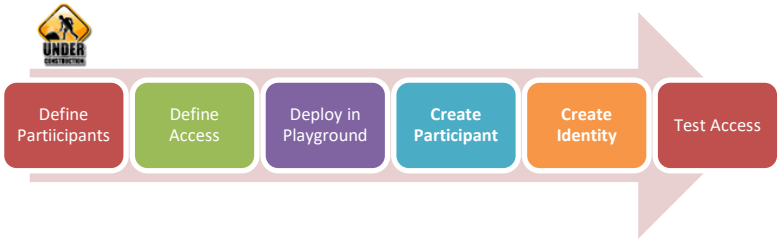
```
namespace org.gryphon.casestudy.college
import org.gryphon.casestudy.base.*
```

Step 2.3: Define ‘College’ Participant



Step 2.4: Define Transactions that ‘College’ Participant can carry out

```
/**  College Specific transactions **/

/**
 * Request affiliation to a University
 */
transaction requestAffiliation {
    o String name
}

/**
 * Enroll new Programs and cources
 */
transaction enrollProgram {
    o String collegeId
    o String programName
}
```



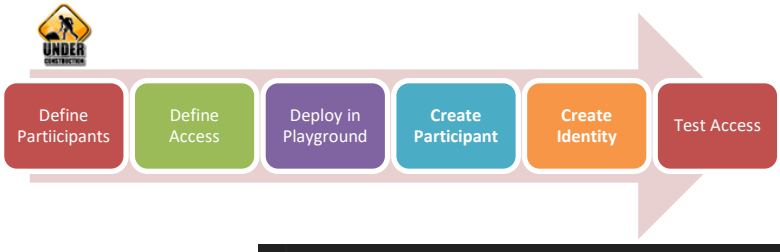
Step 2.5: Similarly Open *‘models/org.gryphon.casestudy.university.cto’* file in Visual Studio Code and define university Participant and Transactions

**** Note**: We have already seen Certificate Asset in previous chapter.

```
/**
 * A University Participant is the governing authority
 */
participant University identified by memberId extends participantBase {
}

/**  University Specific transactions **/

/**
 * Issue a certificate to a student
 * A transaction that can only be executed by the University Participant
 */
transaction issueCertificate {
  o String studentName
  o String programName
}

/**
 * Approve affiliation of College to the University
 * A transaction that can only be executed by the University Participant
 */
transaction approveAffiliation {
    o String memberId
}
```

Step 2.6: Also define student & verifier Participant and Transactions in corresponding .cto model files:

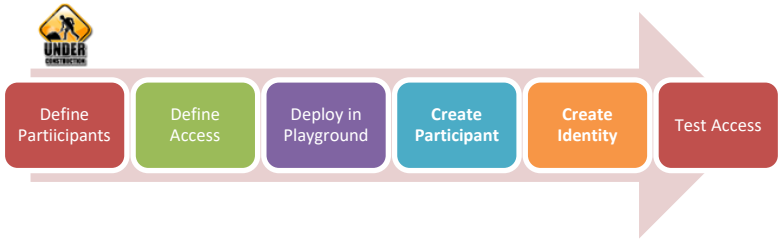
'org.gryphon.casestudy.student.cto'




```
namespace org.gryphon.casestudy.student
import org.gryphon.casestudy.base.*

/**
 * A College Participant that extends the participant base
 */
participant Student identified by memberId extends participantBase {
    o DateTime dob
    o String collegeName
    o String programName
    o String certificateId optional
}

/**  Student Specific transactions **/

/**
 * Student enrolls to a college and program
 */
transaction enrollStudent {
    o String name
    o DateTime dob
    o String collegeName
    o String programName
}
```

'org.gryphon.casestudy.verifier.cto'

```
namespace org.gryphon.casestudy.verifier
import org.gryphon.casestudy.base.*

/**
 * A Verifier who has only readonly attribute to Certificate Asset
 */
participant Verifier identified by memberId extends participantBase {
}

/**  Verifier Specific transactions **/

/**
 * Anyone can verify if the authenticity of the certificate by its ID
 */
transaction verifyCertificate {
    o String certificateId
}
```



Step 3: Define chaincode corresponding to the respective transactions defined.

**** Note:** Writing chaincode was shown in previous chapter. Here we define chaincode for remaining transactions

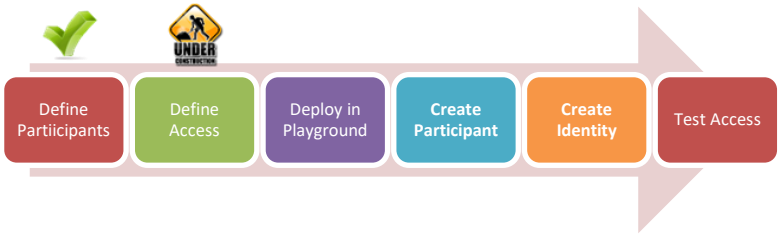
```
 * @transaction
 */
function approveAffiliation(args) {
    var registry;
    return getParticipantRegistry(NS_college + '.College')
        .then(function (assetRegistry) {
```

```
 */
function enrollProgram(args) {
    var registry;
    return getParticipantRegistry(NS_college + '.College')
        .then(function (assetRegistry) {
            registry = assetRegistry;
```

```
 */
function enrollStudent(args) {
    return getParticipantRegistry(NS_student + '.Student')
        .then(function (assetRegistry) {
            var factory = getFactory();
            var studentId = 'Student-' + Date.now().toString();
```

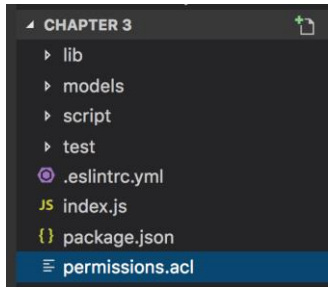
```
 * @transaction
 */
function requestAffiliation(args) {
    return getParticipantRegistry(NS_college + '.College')
        .then(function (assetRegistry) {
            var factory = getFactory();
            var collegeId = 'College-' + Date.now().toString();
```

Task 1 is complete!



TASK#2: Defining Access Control

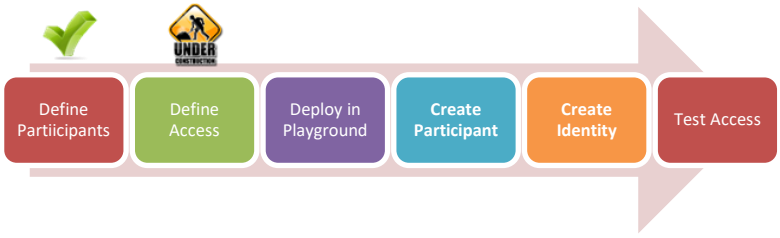
Step 1: Open 'Chapter 3' code provided in Visual Studio Code and Open *'permissions.acl'* file



Step 1: Define permissions for NetworkAdmin User.

***#Info:** If '.acl' is NOT defined all users have full permissions, however once '.acl' is defined all permissions are revoked and you need to define permissions even for the NetworkAdmin in order for the Admin to perform deployment and administrative functions.*

Step 1.1: Grant business network administrators full access to user resources



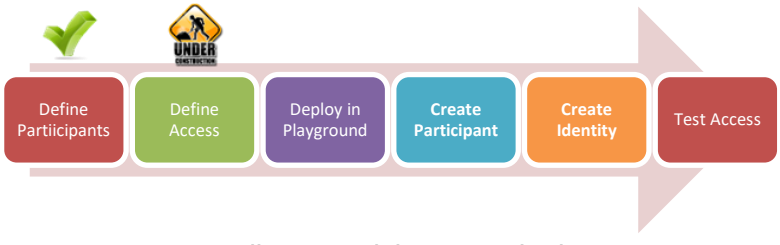
Step 1.2: Grant business network administrators full access to system resources

```
rule NetworkAdminSystem {
    description: "Grant business network administrators full access to system
    resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}
```

Step 2: Now that the NetworkAdmin access to all system and user resources are provided, we provide access to the Participants in our example. The intension here is to provide isolation, so that the Participant can carry out operations / transactions related to their authority only. All other operations/transactions should be restricted.

Step 2.1: Grant College Participant full access to resources in college namespace and restrict access to resources in other namespace

```
/*** Define College Participant Access ***/
rule CollegeResourceAccess {
    description: "Grant College Participant full access to resources in
    college namespace"
    participant: "org.gryphon.casestudy.college.College"
    operation: ALL
    resource: "org.gryphon.casestudy.college.**"
    action: ALLOW
}
```

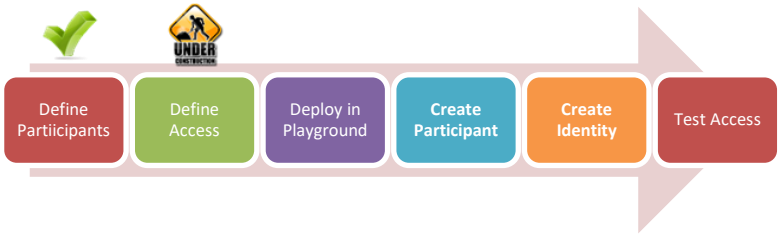


Step 2.2: Grant College Participant readonly access to system resources. This will ensure we are able to use Playground to read status of all resources and we can test our ACL file

```
/*** Required for Playgroud to view all resources ***/
rule CollegeSystemReadOnly {
    description: "Grant College Particpant readonly access to system
    resources"
    participant: "org.gryphon.casestudy.college.College"
    operation: READ
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}
```

Step 2.3: Grant access to perform transactions. Any transaction will create an entry in the historian, as transaction record is immutable. We need to provide this access to our College Participant so that it can perform appropriate transactions '*requestAffiliation'* and '*enrollProgram'*.

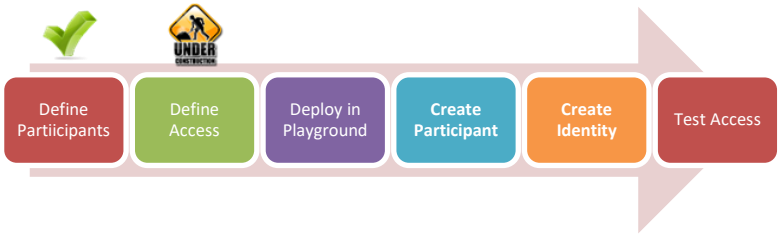
```
/*** Required to execute any Transaction ***/
rule CollegeSystemHistorianCreate {
    description: "Grant access to perform transactions"
    participant: "org.gryphon.casestudy.college.College"
    operation: CREATE
    resource: "org.hyperledger.composer.system.HistorianRecord"
    action: ALLOW
}
```



Step 3: Similarly we define access to other Participants:

Step 3.1: For Student Participant

```
/*** Define Student Participant Access ***/
rule StudentResourceAccess {
    description: "Grant Student Participant full access to resources in
    college namespace"
    participant: "org.gryphon.casestudy.student.Student"
    operation: ALL
    resource: "org.gryphon.casestudy.college.**"
    action: ALLOW
}

/*** Required for Playgroud to view all resources ***/
rule StudentSystemReadOnly {
    description: "Grant Student Participant readonly access to system
    resources"
    participant: "org.gryphon.casestudy.student.Student"
    operation: READ
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}

/*** Required to execute any Transaction ***/
rule StudentSystemHistorianCreate {
    description: "Grant access to perform transactions"
    participant: "org.gryphon.casestudy.student.Student"
    operation: CREATE
    resource: "org.hyperledger.composer.system.HistorianRecord"
    action: ALLOW
}
```



Step 3.1: For University Participant, we need to update College Registry and also Student Registry. Hence for simplicity we will provide them access to all user resources. In actual we should explicitly define what operations, on what resources needs to be granted.

```
/*** Define University  Access ***/
rule UniversityResourceAccess {
    description: "Grant University Participant full access to resources in
    all namespaces"
    participant: "org.gryphon.casestudy.university.University"
    operation: ALL
    resource: "org.gryphon.casestudy.**"
    action: ALLOW
}

/*** Required for Playgroud to view all resources ***/
rule UniversitySystemReadOnly {
    description: "Grant University Participant readonly access to system
    resources"
    participant: "org.gryphon.casestudy.university.University"
    operation: READ
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}

/*** Required to execute any Transaction ***/
rule UniversitySystemHistorianCreate {
    description: "Grant access to perform transactions"
    participant: "org.gryphon.casestudy.university.University"
    operation: CREATE
    resource: "org.hyperledger.composer.system.HistorianRecord"
    action: ALLOW
}
```

Task 2 is complete!

TASK#3: Deploy in Playground

Step 1: Open ‘Chapter 3’ code provided in Visual Studio Code. Open terminal window and cd into the script’ directory

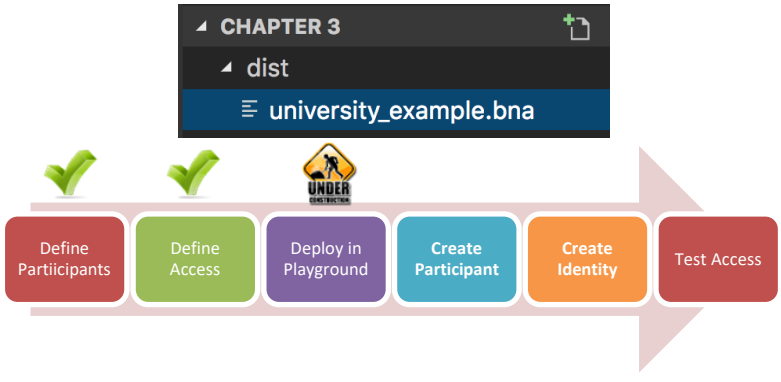
```
cd script
```

Step 2: Create Business Network Archive using the provided scripts; Type the following command in terminal window to run the archive script

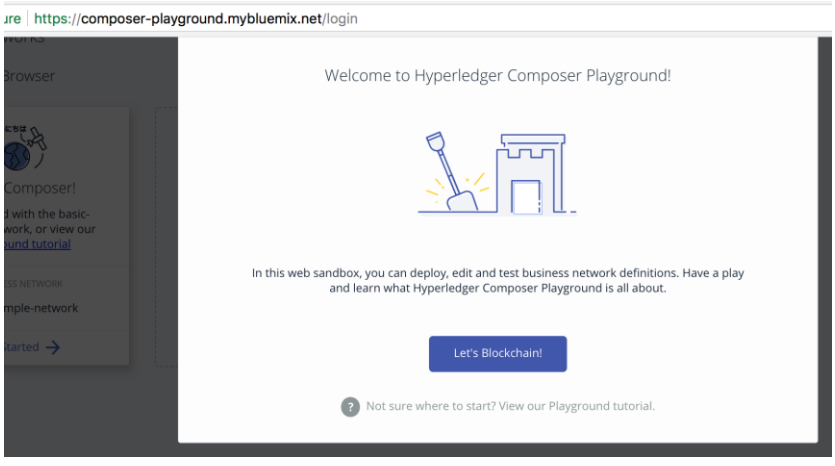
```
./createArchive.sh
```

```
Creating Business Network Archive

Looking for package.json of Business Network Definition
        Input directory:                                    /Chapter 3

Found:
        Description: Creating models for our university example
        Name: university_example
        Identifier: university_example@0.0.1

Written Business Network Definition Archive file to
        Output file: ./dist/university_example.bna

Command succeeded
```

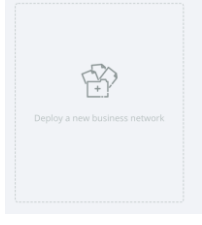
On successful completion of the command a *‘university_example.bna’* is created in the ‘dist’ folder



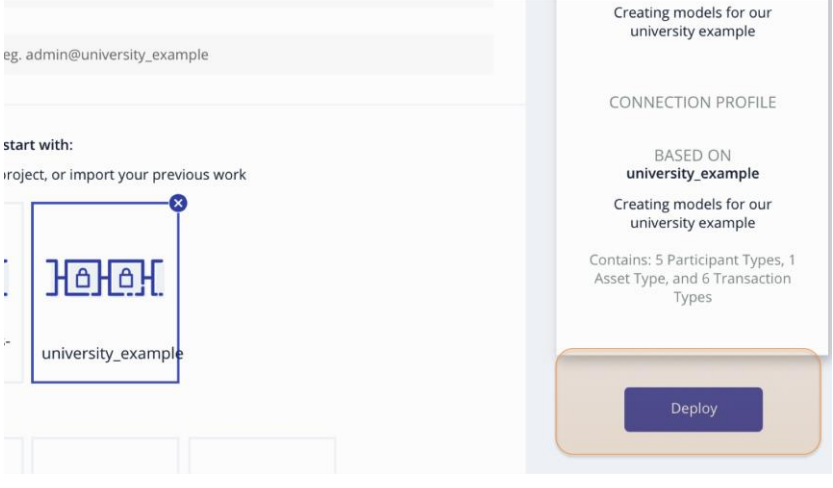
Step 3: Launch internet browser and goto the following URL:
https://composer-playground.mybluemix.net



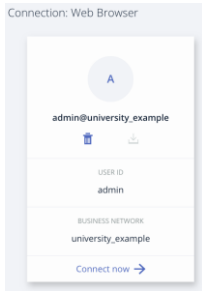
Step 4: Click 'Deploy Network' as below



Step 5: Choose 'Drop File to Upload' option and upload the recently created ***'university_example.bna'*** file from the ***'dist'*** folder.



Step 6: Choose ‘Deploy’ button to deploy the ***‘.bna’*** file

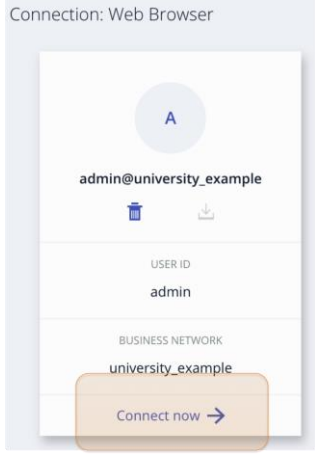


NetworkAdmin User is created on deployment of business network.

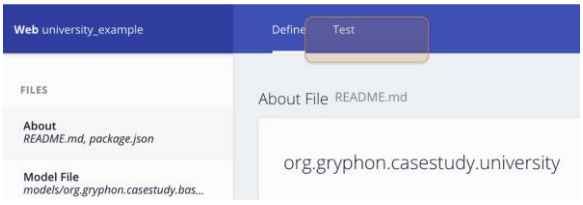
Task 3 is complete!

TASK#4: Create Participants

Step 1: Business Network Archive was deployed in previous task to Playground. Use the NetworkAdmin User created on deployment to connect to the network.

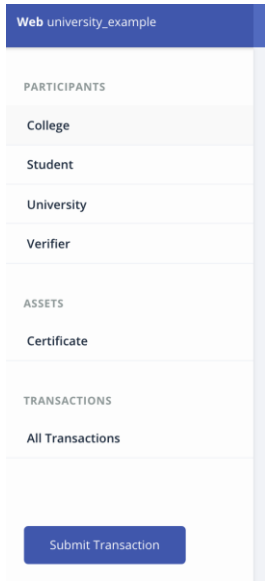


Step 2: Click on ‘Test’ tab to start interacting with the business network

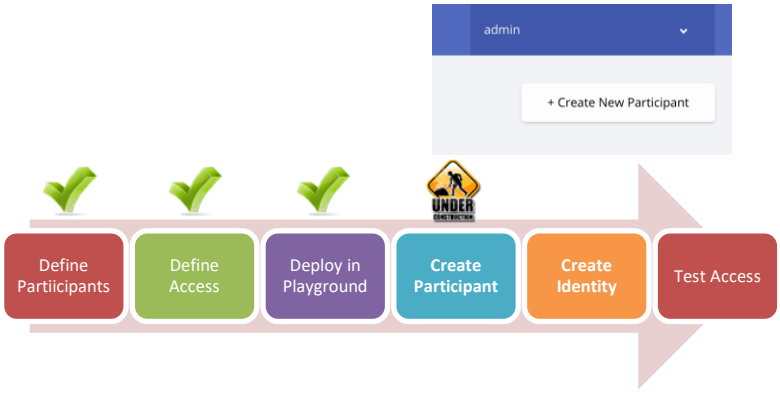

Web university_example
Define
Test
FILES
About
README.md, package.json
Model File
models/org.gryphon.casestudy.bas...
About File README.md
org.gryphon.casestudy.university

Step 3: Creating College participants:

Step 3.1: To create College Participant, click on ‘College’ Tab under participant list



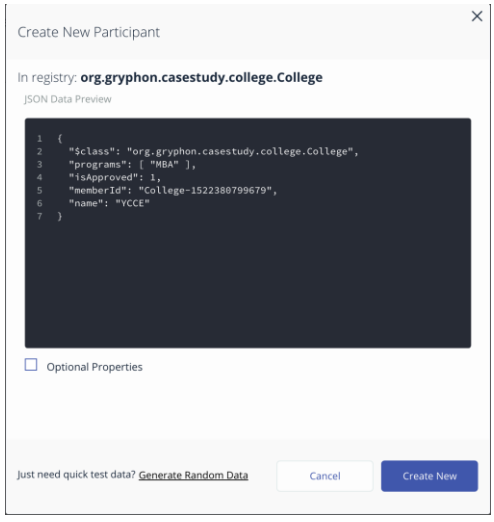
Step 3.2: Then use ‘Create new Participant’ button on the top right to add participant details



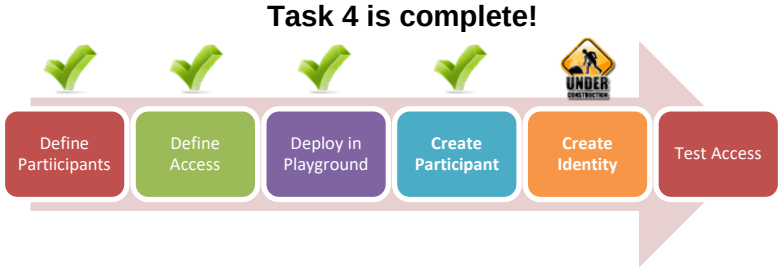
Step 3.3: Use following details to create a new participant

```
{
```

```
  "$class": "org.gryphon.casestudy.college.College",
  "programs": [ "MBA" ],
  "isApproved": 1,
  "memberId": "College-1522380799679",
  "name": "YCCE"
}
```

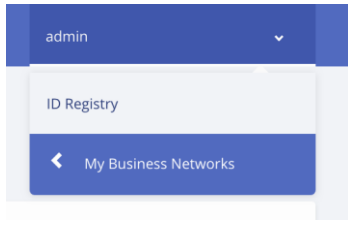


On completion a new College Participant is created.

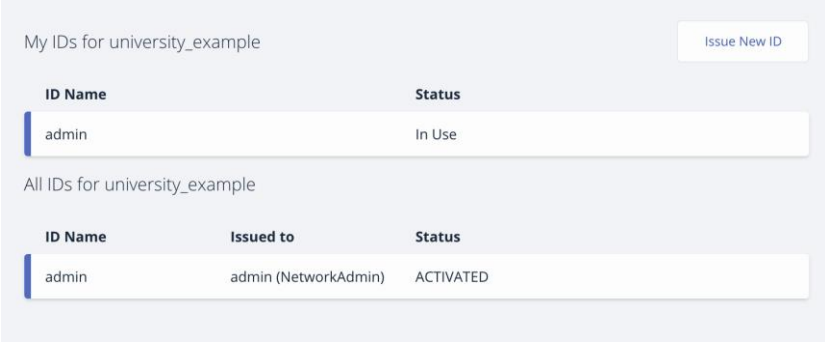


TASK#5: Create Identity

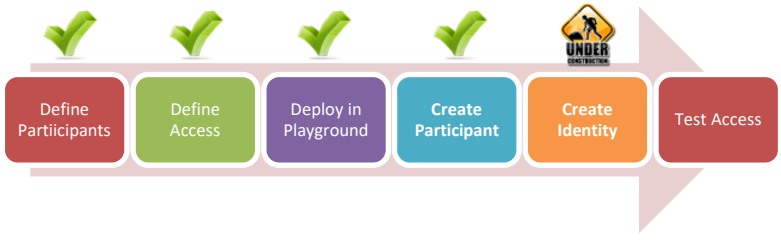
Step 1: Business Network Archive was deployed and 'College' Participant was created in previous task. Use the dropdown button on top right with the admin label and click on 'ID Registry'

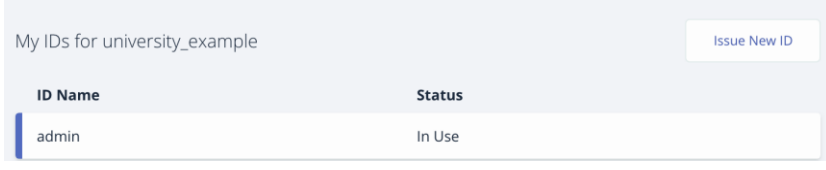


‘ID Registry’ contains the entries of all the Participants and associated Identities. New Identities for the participant can be created or earlier identities can be revoked from here.

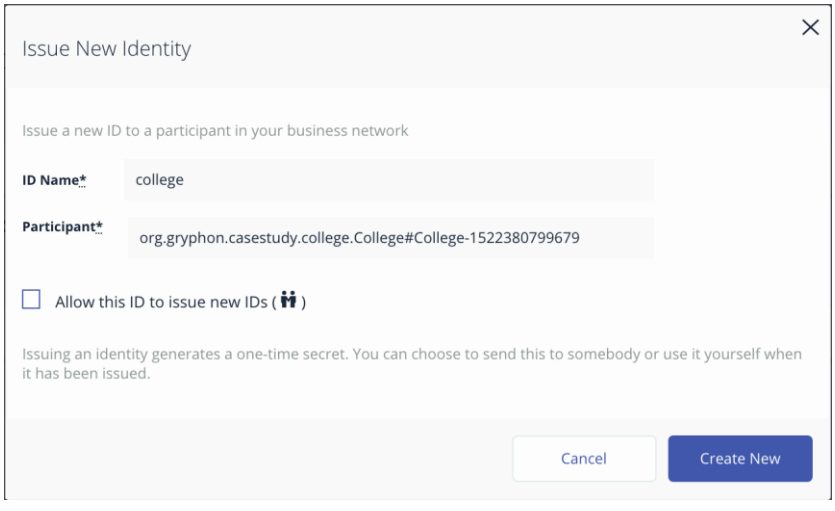


Step 2: Click on ‘Issue New ID’

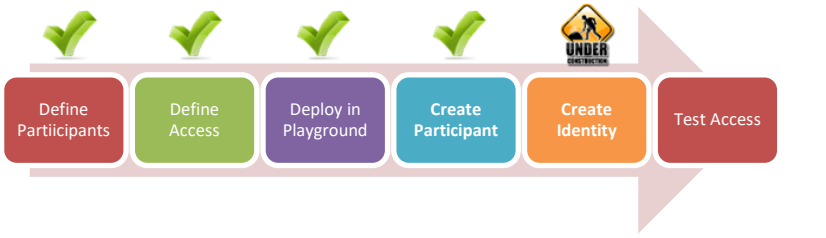


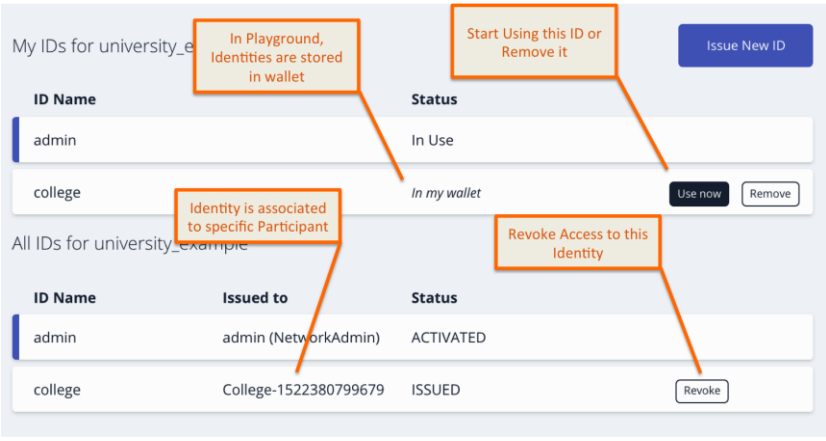


Step 3: Enter a name to the identity and specify to which Participant identity has to be issued. In our case a college participant with ID: ***College-1522380799679***



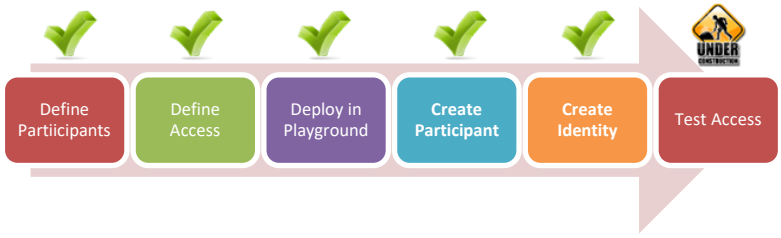
A new Identity has been created for the College Participant. Listing as shown below;





Step 4: To use the newly created identity for College participant, click “Use now” button against the identity.

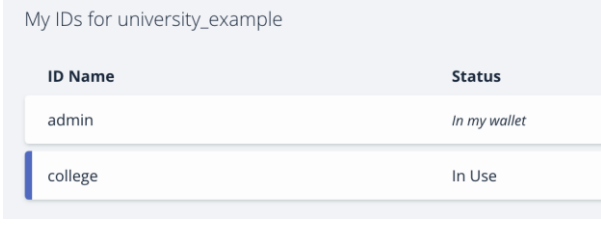
Task 5 is complete!



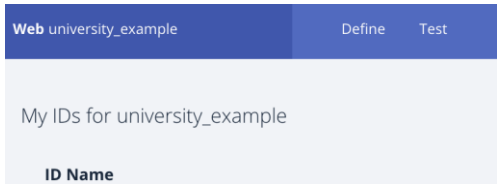
TASK#6: Test Access

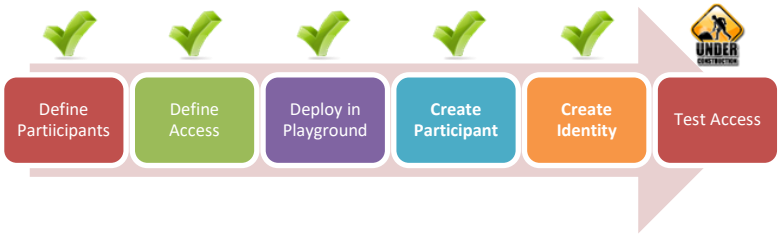
Step 1: Business Network Archive was deployed, 'College' Participant and its associated identity was created in previous task. We need to use this newly created Identity to test the access. Click 'Use now'.

'admin' ID's status will start showing as 'In my wallet' and college ID will be 'In Use', indicating that our new Identity is at work.



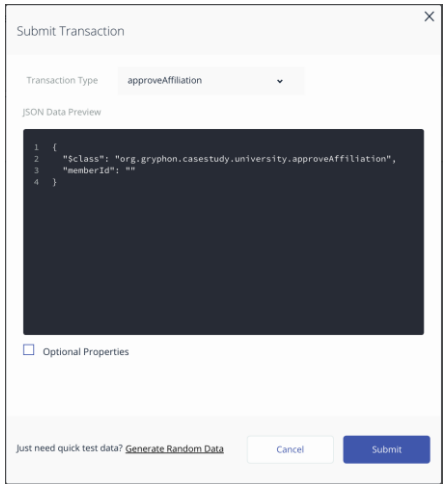
Step 2: Click on the 'Test' tab to navigate to the Playground





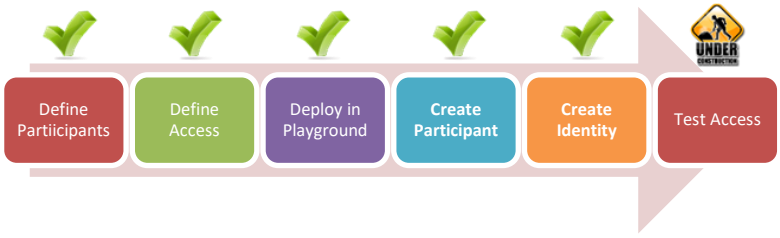
Step 3: We can invoke transaction using the 'Submit Transaction' button on left bottom of the 'Test' page

On clicking the button, transaction dialog shall appear;

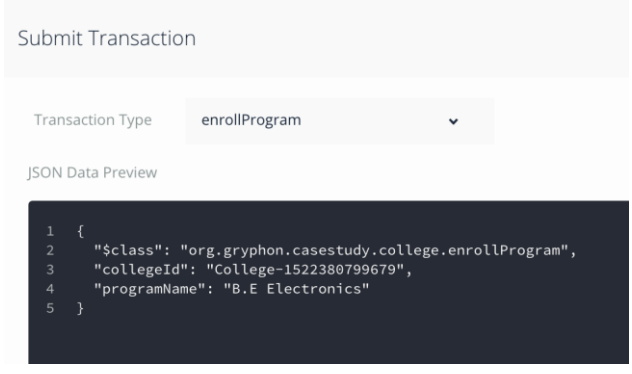


Step 4: Testing a **permissioned** transactions [***'requestAffliation'*** & ***'enrollProgram'***]

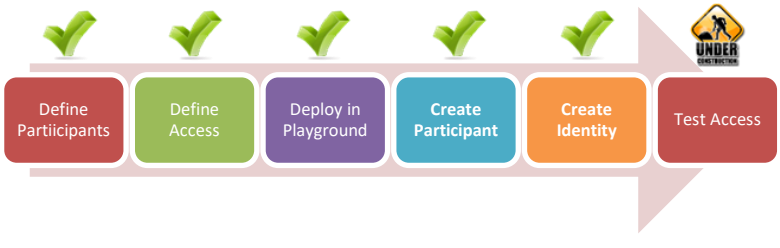
Step 4.1: In submit transaction dialog select transaction type as ***'enrollProgram'*** and use the following json to add "B.E Electronics" program to the existing college 'YCCE' with ID: College-1522380799679




```
{
  "$class": "org.gryphon.casestudy.college.enrollProgram",
  "collegeId": "College-1522380799679",
  "programName": "B.E Electronics"
}
```



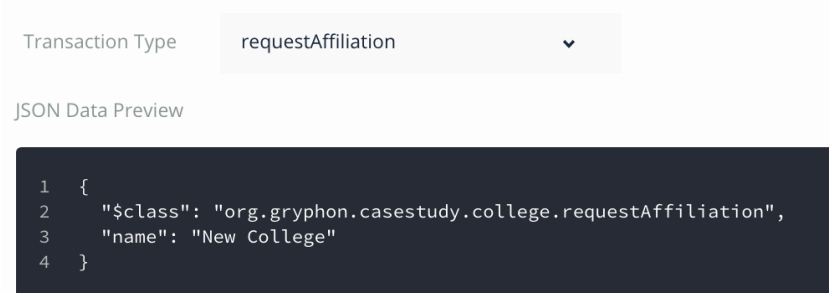
Step 4.2: Click Submit. On successful completion, we should be able to see a second program ‘B.E Electronics’ in the list of programs for the specific college



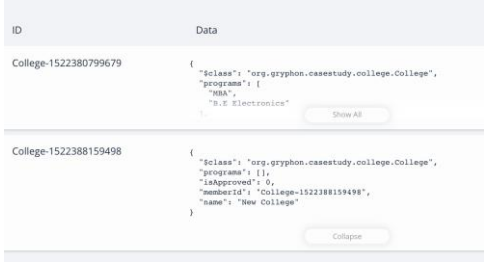
Step 4.3: Similarly Submit a new ***‘requestAffliation’*** transaction using argument data as below;

{

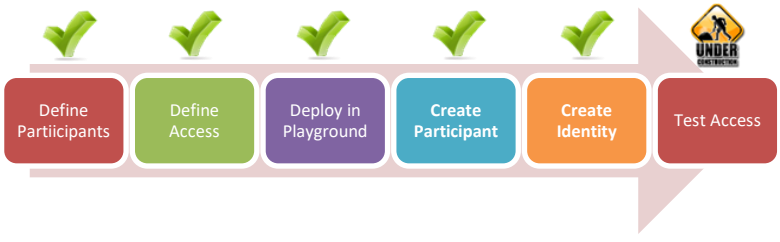
```
  "$class": "org.gryphon.casestudy.college.requestAffiliation",
  "name": "New College"
}
```



On successful execution a new college participant will be created.



#Result: *We are able to successfully execute chaincode transactions within namespace org.gryphon.casestudy.college and access as specified in 'permissions.acl' file*

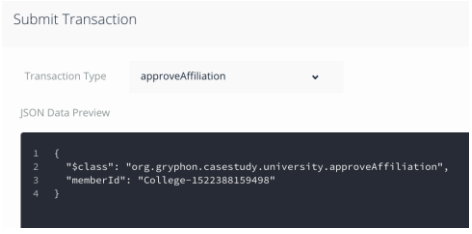


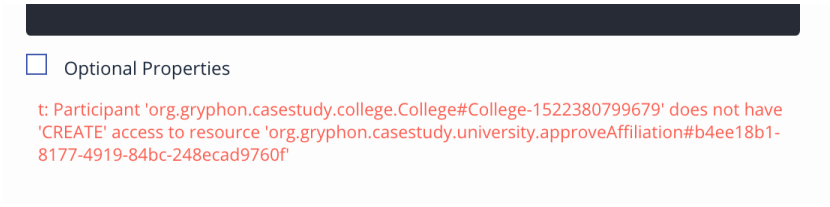
Step 5: Testing a **restricted permissioned** transactions [***'approveAffliation'*** & ***'issueCertificate'***]

Step 5.1: Let us try to execute ***'approveAffliation'*** transaction which only University is authorized as per our 'permissions.acl' file and see if permission are granted as expected;

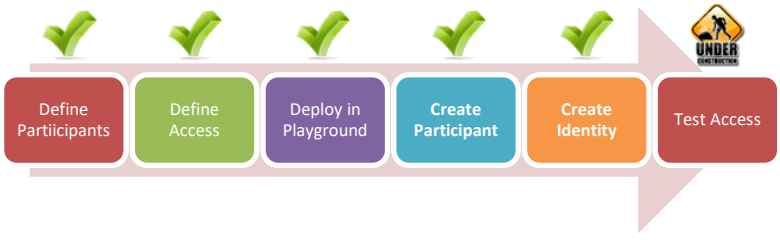
Lets approve 'New College' participant;

***** Note:*** *Please use college ID that is generated by your system for 'New College' participant*

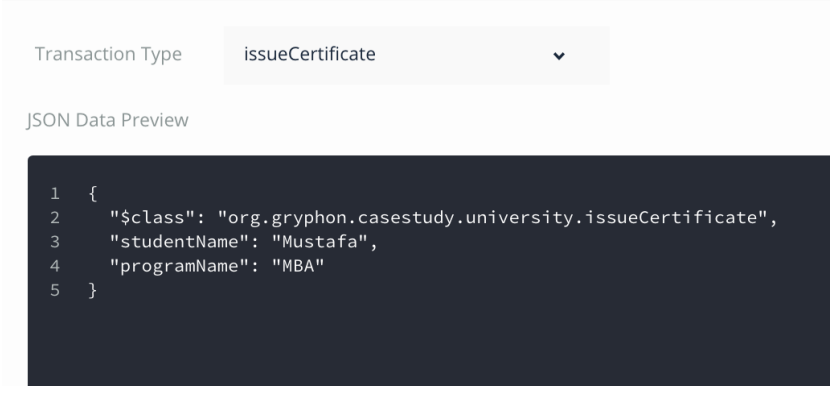




Transaction fails, showing that college participant does not have appropriate permissions



Step 5.2: Similarly Let us try to execute ***'issueCertificate'*** transaction which only University is authorized as per our 'permissions.acl' file and validate our permissions;



On submission of transaction we get permissions error as below;

#Result: *College Participant access to chaincode transactions are restricted only within namespace org.gryphon.casestudy.college. Hyperledger is a private permissioned network and access to the network can be provided/revoked using identities and selective permissions are defined in '.acl' file.*

Task 6 is complete!

Test Your Knowledge

In Hyperledger Fabric business network application, access is granted to both known and unknown entities?
- TRUE
- FALSE

In which file do you define access permissions?
- .ACL file
- .CTO file
- .json file
- .yml file

Access to Hyperledger Fabric Business Network Application is granted based on the _____ of the participant
- Role
- Identity
- Documents

Participants are assigned an identity using:
- User Name / Password
- X509 Certificate
- SSO

Apart from Participants, are Organization and Hyperledger Infrastructure also issued a X509 certificate?
- True
- False

SUMMARY

Hyperledger is a permissioned network. To be able to access resources or perform transactions network business cards/identities are needed.

In this chapter we have learnt the following:

- Participant & and their identities
- Adding a new participant to a participant registry
- Issuing a new identity to a participant
- Binding an existing identity to a participant
- Defining permissions to participant users using ACL (Access control Language)

CHAPTER 6: HYPERLEDGER – CLIENT APP

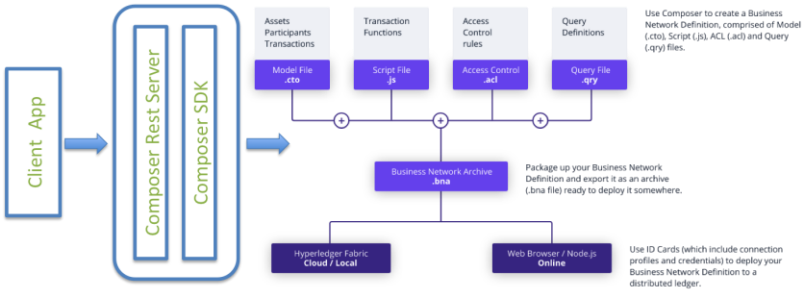
Theory

This chapter will be more hands-on. It will cover how we can build the client app, which can interaction with the fundamental blocks of Hyperledger blockchain i.e;

- Connecting to the Business Network
- Access Participant and Asset registries
- Invoke Transactions
- Query Resources (Static & Dynamic Queries)
- Handling events

High Level Architecture

Model file, Transaction functions (chaincode), access control file and the static query file make the Business Network Archive (Package) for the Hyperledger Fabric.



We will use Node.js and composer SDK to connect to the Fabric network and access/interaction with the blockchain resources.

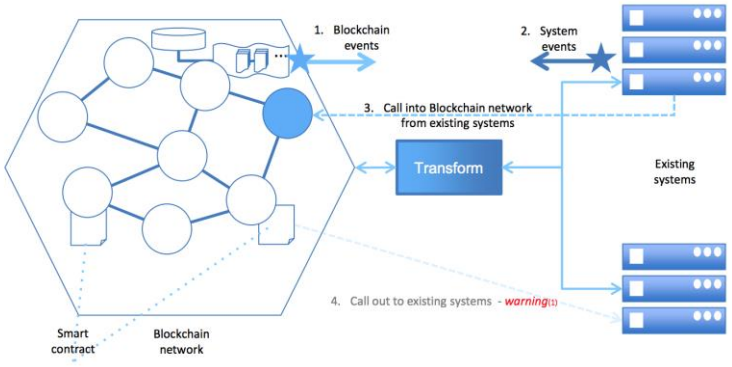
Queries:

The native query language can filter results returned using criteria and can be invoked in transactions to perform operations, such as updating or removing assets on result sets.

Queries are defined in a query file (.qry) in the parent directory of the business network definition. Queries contain a WHERE clause, which defines the criteria by which assets or participants are selected.

Events

Event creates notifications of significant operations on the Blockchain (e.g. a new block), as well as notifications related to a milestone achieved while processing a smart contract/chaincode. Does not include event distribution.



The client app can subscribe to this event and take appropriate business actions. It's an important part of any system to provide an insight ask for immediate attentions.

In the lab section we will generate an event in the chaincode and client app will subscribe for this event. We can then utilize this event to either send an email or appropriately notify the participant.

Lab 4: Coding client App, Queries & Events

Tasks

- Coding Client App
  - requestAffiliation()
  - getCollegeList()
  - approveAffiliation()
  - enrollProgram()
  - takeAdmission()
  - getStudentList()
  - issueCertificate()
  - verifyCertificate()
- Querying Registries
- Events & Subscription

TASK#1: Coding Client App

Step 1: Open 'Chapter 4' code provided in Visual Studio Code.

*#**Info**: 'Chapter 4' is a Business Network Scaffolding created as in Chapter#2-Task 1 with following changes:*

1. *Chaincode 'logic.js' is replaced with 'chaincode.js'*
2. *Test 'logic.js' is replaced with 'chaincode-test.js'*
3. *'script' folder is added*
4. *'permissions.acl' added*
5. *'client_app' folder added*

Step 2: Open terminal window of Visual Studio code and type run the following command

```
npm install
```

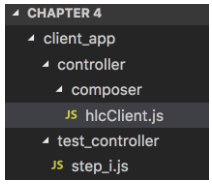
Step 3: Cd in the scripts folder and run buildAndDeploy.sh script to deploy the business network [Models and chaincode]

```
cd script
./buildAndDeploy.sh
```

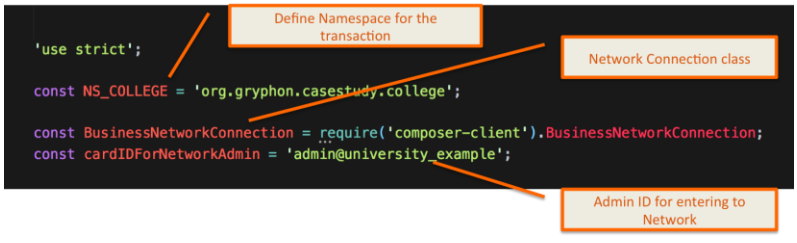
TASK#1.1: Coding requestAffiliation()

Step 1: Browse to directory ‘client_app/controller/composer’ and open file ‘hlcClient.js

,

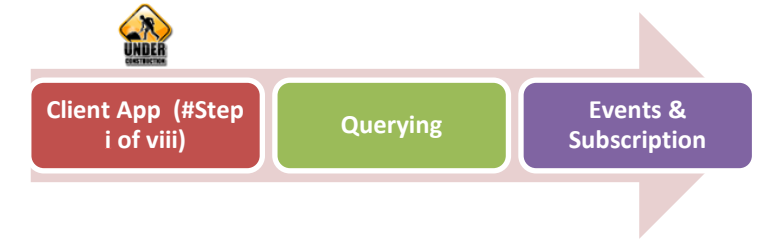


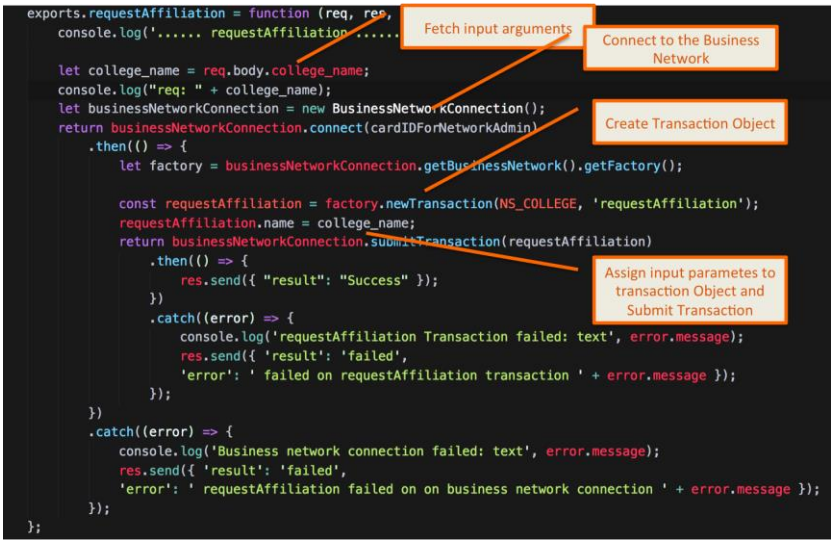
Step 2: Coding requestAffiliation() method



Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Create Transaction Object
- Set Transaction Arguments
- Submit Transaction
- Handle failure conditions by logging error info






```
exports.requestAffiliation = function (req, res,
    console.log('..... requestAffiliation .....

    let college_name = req.body.college_name;
    console.log("req: " + college_name);
    let businessNetworkConnection = new BusinessNetworkConnection();
    return businessNetworkConnection.connect(cardIDForNetworkAdmin)
        .then(() => {
            let factory = businessNetworkConnection.getBusinessNetwork().getFactory();

            const requestAffiliation = factory.newTransaction(NS_COLLEGE, 'requestAffiliation');
            requestAffiliation.name = college_name;
            return businessNetworkConnection.submitTransaction(requestAffiliation)
                .then(() => {
                    res.send({ "result": "Success" });
                })
                .catch((error) => {
                    console.log('requestAffiliation Transaction failed: text', error.message);
                    res.send({ 'result': 'failed',
                    'error': ' failed on requestAffiliation transaction ' + error.message });
                });
        })
        .catch((error) => {
            console.log('Business network connection failed: text', error.message);
            res.send({ 'result': 'failed',
            'error': ' requestAffiliation failed on on business network connection ' + error.message });
        });
};
```

Step 3: To validate if requestAffiliation() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 4: Run the test script using nodejs as below

```
node step_i.js
```

The result success signifies that our transaction was successful

```
..... requestAffiliation .....
req: #CollegeNAME
{ result: 'Success' }
```

Task 1 #i is complete!

TASK#1.2: Coding getCollegeList()

Step 1: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js

```
exports.getCollegeList = function (req, res, next) {
    console.log('..... College List .......');
    let allColleges = new Array();
    let businessNetworkConnection = new BusinessNetworkConnection();
    return businessNetworkConnection.connect(cardIDForNetworkAdmin)
        .then(() => {
            return businessNetworkConnection.getParticipantRegistry(NS_COLLEGE + '.College')
                .then(function (registry) {
                    return registry.getAll()
                        .then((collegeList) => {
                            for (let each in collegeList) {
                                (function (_idx, _arr) {
                                    let _jsn = _arr[_idx];
                                    let jsn = { "id": _jsn.memberId,
                                    "name": _jsn.name,
                                    "is_approved": _jsn.isApproved,
                                    "programs": _jsn.programs };
                                    allColleges.push(jsn);
                                })(each, collegeList);
                            }
                            res.send({ 'result': 'success', 'college_lis
                        })
                        .catch((error) => {
                            console.log('error with getAll Colleges', error);
                            res.send({ 'result': 'false', 'college_list': [] });
                        });
                })
                .catch((error) => {
                    console.log('error with getCollegeList', error);
                    res.send({ 'result': 'false', 'college_list': [] });
                });
        })
        .catch((error) => { console.log('error with business network Connect', error); });
```

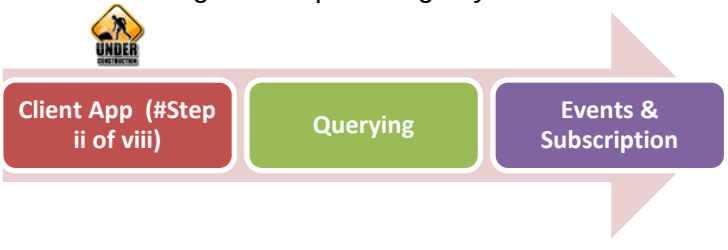
Connect to the Business Network

Get College Participant Registry

Contruct json array for return

Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Get College Participant Registry



- Get All participants
- On success, build a json and return
- On error Handle failure conditions by logging error info

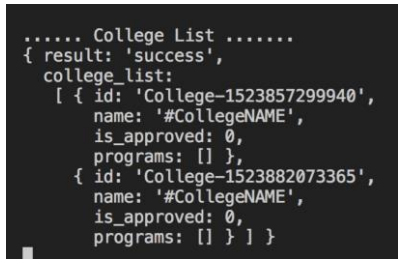
Step 2: To validate if getCollegeList() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 3: Run the test script using nodejs as below

```
node step_ii.js
```

The result success with list of colleges signifies that our transaction was successful



Task 1 #ii is complete!

TASK#1.3: Coding approveAffiliation()

Step 1: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js

Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Create Transaction & assign input arguments
- Handle failure conditions by logging error info

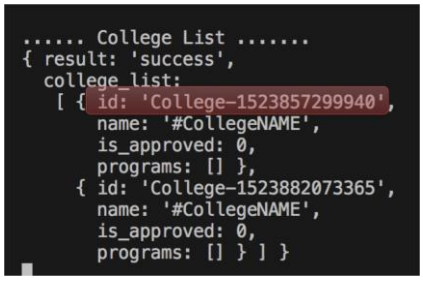
Step 2: To validate if approveAffiliation() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 3: Run the test script using nodejs as below to FETCH list of all colleges

```
node step_iii.js
```

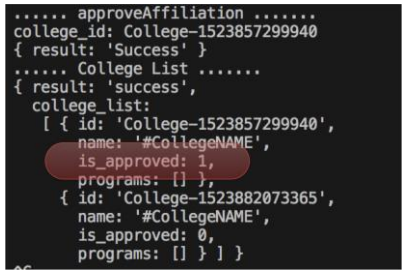
The result success with list of colleges signifies that our transaction was successful



Step 4: Note the **CollegeID** in red above and run the test below script using nodejs to approve affiliation of college

```
node step_iii.js ReplaceWithYourCollegeID
```

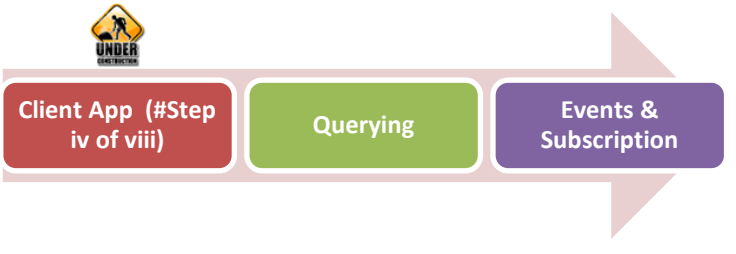
The result success with approved flag of specified college ID signifies that our transaction was successful



Task 1 #iii is complete!

TASK#1.4: Coding enrollProgram()

Step 1: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js



Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Create Transaction & assign input arguments

- Handle failure conditions by logging error info

Step 2: To validate if enrollProgram() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 3: Run the test script using nodejs as below
**Note: Use CollegeID that was generated in you Use Case

```
college_list:
 [ { id: 'College-1523857299940',
     name: '#CollegeNAME',
```

```
node step_iv.js College-1523857299940 M.B.A
```

The result success with list of program for the specified college signifies that our transaction was successful

```
...... enrollProgram .......
college_id: College-1523948141134
program_name: MBA
{ result: 'Success' }
...... College List .......
Programs Enrolled for College-1523948141134: MBA
```

Task 1 #iv is complete!

TASK#1.5: Coding takeAdmission()

Step 1: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js

Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Create Transaction & assign input arguments
- Handle failure conditions by logging error info

Step 2: To validate if takeAdmission() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 3: Run the test script using nodejs as below

**Note: In actual scenario ID for Student, College and Program Name should be taken. However this is just a demo Use Case hence we are simply taking names

```
node step_v.js "John Doe" CollegeNAME MBA
```

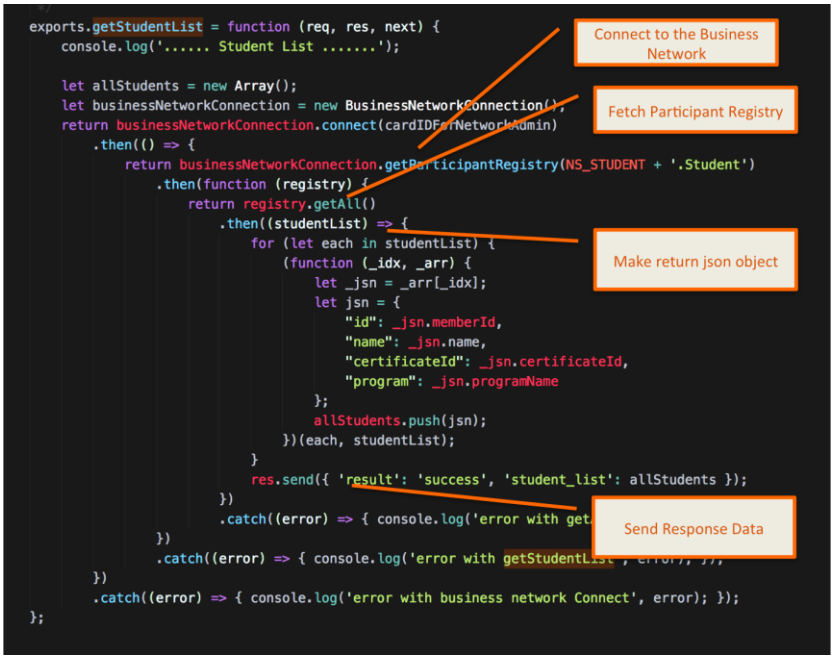
The result success signifies that our transaction was successful

```
  student_name: John Doe
  student_dob: Tue Apr 17 2018 12:30:36 GMT+0530 (IST)
  college_name: CollegeNAME
  program_name: MBA
  { result: 'Success' }
   enrollStudent Transaction Success......
```

Task 1 #v is complete!

TASK#1.6: Coding getStudentList()

Step 1: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js



Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Create Transaction & assign input arguments
- Handle failure conditions by logging error info

Step 2: To validate if getStudentList() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 3: Run the test script using nodejs as below

```
node step_vi.js
```

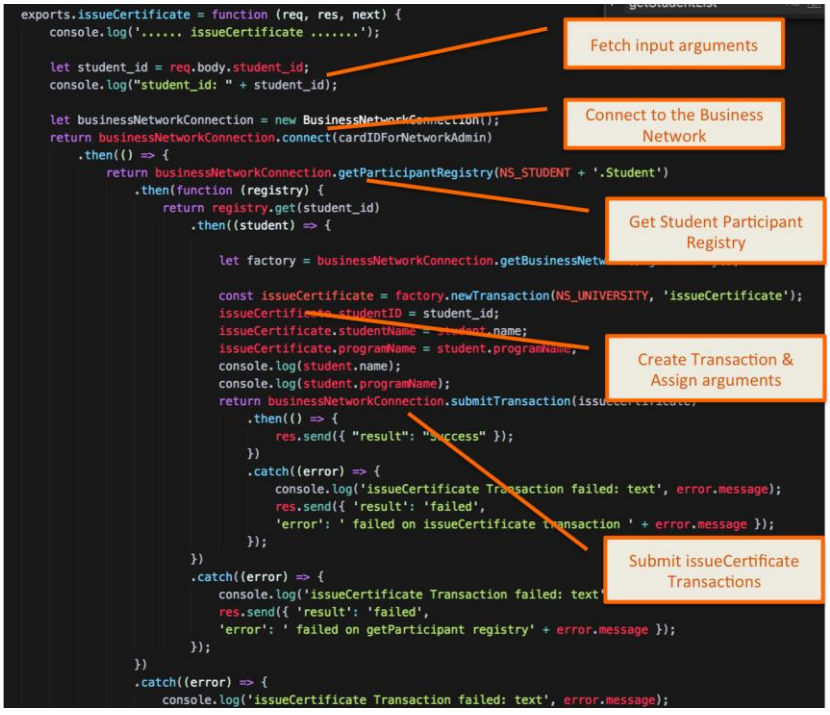
The result success signifies that our transaction was successful

```
..... Student List .......
{ result: 'success',
  student_list:
   [ { id: 'Student-1523948446529',
       name: 'John Doe',
       certificateId: undefined,
       program: 'MBA' } ] }
```

Task 1 #vi is complete!

TASK#1.7: coding issueCertificate()

Step 1: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js



Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Fetch Student Participant Registry
- Get Student Records
- Create issueCertificate Transaction & assign input arguments
- Handle failure conditions by logging error info

Step 2: To validate if issueCertificate() works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 3: Run the test script using nodejs as below

****Note**: Please use Student ID generated on your system

```
...... Student List .......
{ result: 'success',
  student_list:
   [ { id: 'Student-1523948446529',
       name: 'John Doe',
       certificateId: undefined,
       program: 'MBA' } ] }
```

```
node step_vii.js Student-1523948446529
```

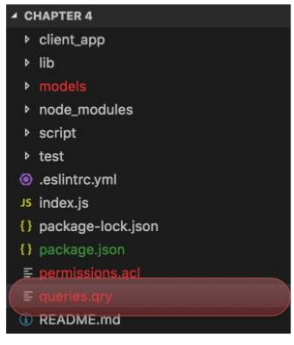
The result success signifies that our transaction was successful

```
...... issueCertificate .......
student_id: Student-1523948446529
John Doe
MBA
{ result: 'Success' }
...... Student List .......
{ result: 'success',
  student_list:
   [ { id: 'Student-1523948446529',
       name: 'John Doe',
       certificateId: 'CertificateID-1523948684198',
       program: 'MBA' } ] }
```

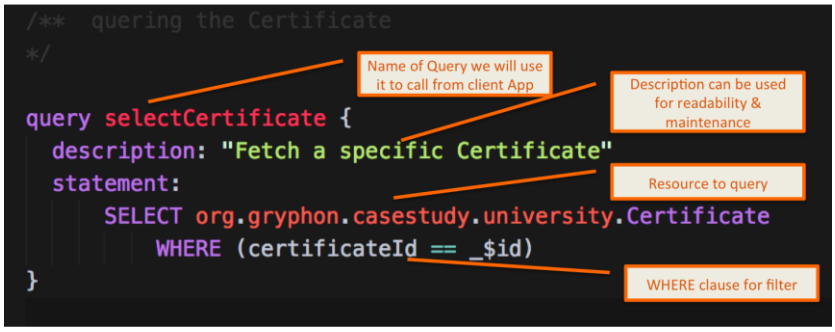
Task 1 #vii is complete!

TASK#2: Using Queries

Step 1: Open ‘Chapter 4’ code provided in Visual Studio Code.
Step 2: Open file ‘queries.qry’

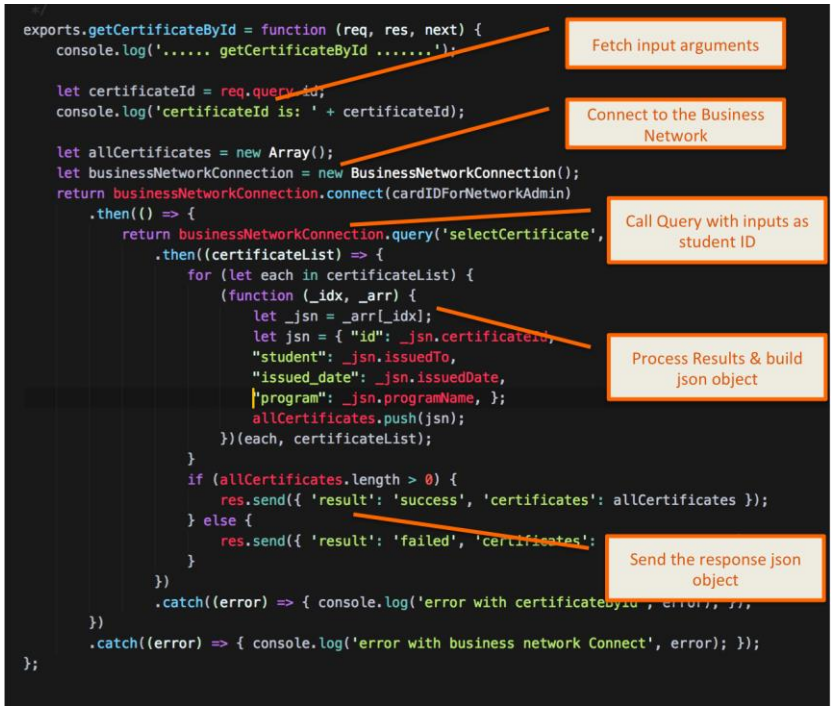


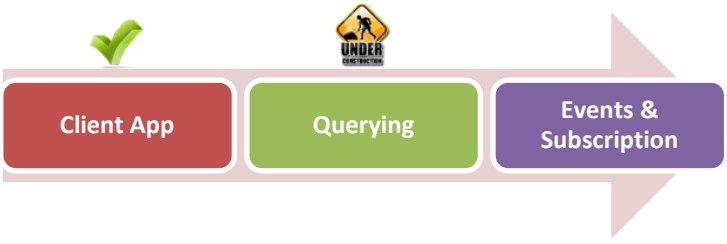
Step 3: Define query to find student by its ID



Step 4: Browse to directory 'client_app/controller/composer' and open file 'hlcClient.js

Step 5: Coding getCertificateById() client App method





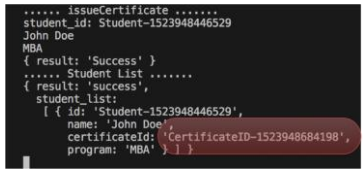
Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Call the query using student ID as filter
- Build the response json object
- Handle failure conditions by logging error info

Step 6: To validate if getCertificateById () works, open terminal in visual studio code and cd into following directory

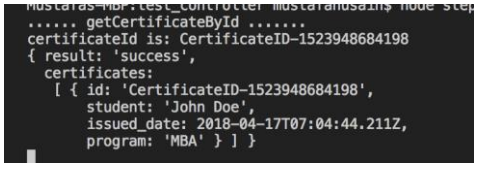
```
cd client_app/test_controller/
```

Step 7: Run the test script using nodejs as below
****Note**: Please use Certificate ID generated on your system




```
node step_viii.js CertificateID-1523948684198
```

The result success with details of certificate signifies that our transaction was successful

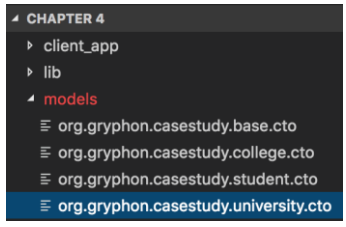


Task 2 is complete!

TASK#3: Blockchain Events & Subscription

Step 1: Open ‘Chapter 4’ code provided in Visual Studio Code.

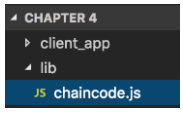
Step 2: Open file ‘org.gryphon.casestudy.university.cto’ under models folder



Step 3: Define Event

```
/**
 * Notify Issue of certificate
 * An event that can be substribed
 */
event certificateIssuedEvent {
  o String certificateId
}
```

Step 4: Browse to directory ‘lib’ and open file ‘chaincode.js



Step 5: Add Emit Event to the issueCertificate() chaincode

```
var event = factory.newEvent(NS_UNIVERSITY, 'certificateIssuedEvent');
event.certificateId = certificateId;
emit(event);
```

Step 6: Browse to directory ‘client_app/controller/composer’ and open file ‘hlcClient.js

Step 7: Subscribing to Events. Coding subscribeForEvents() method

```
let bRegistered = false;
exports.subscribeForEvents = function (req, res, next)
{
   console.log('..... subscribeForEvents .......');

   if (bRegistered) { res.send('Already Registered'); }
   else{
       bRegistered = true;
       let businessNetworkConnection = new BusinessNetworkConnection();
       businessNetworkConnection.setMaxListeners(50);
       return businessNetworkConnection.connect(cardIDForNetworkAdmin)
       .then(() => {
           // using the businessNetworkConnection, start monitor
           // when an event is provided, call the _monitor funct
           // the al_connection, f_connection and event information
           businessNetworkConnection.on('event', (event) => {_monitor(event); });
           res.send('event registration complete');
       }).catch((error) => {
           // if an error is encountered, log the error and send
           console.log(method+' business network connection faile
           res.send(method+' business network connection failed'+error.message);
       });
   }
};
```

Check if already registered

Connect to the Business Network

Set Listner Limits

Subscribe to Event

Actions:

- Create Business Network object
- Connect to network using permissioned card name (Participant or Admin) depending on Use Case
- Set connection limits
- Subscribe to the events
- Handle failure conditions by logging error info

Step 8: Any blockchain event emitted will now call ***'_monitor()'***

```
function _monitor(_event)
{
    let method = '_monitor';
    console.log(method+ ' _event received: '+_event.$type);

    // Can use Socket.io to connect to UI
    // io.emit(_event.name, _event.value);

    if(_event.$type != 'certificateIssuedEvent') {
        // Probable Action: Fetch Certifcate & Student Details and Send Email
        // sendEmail(subject, message);
    }
}
```

- For all events same_monitor() method will be called

- We need to check if event called was ‘certificateIssueEvent’, we do it by peeking into _event.$type variable

Step 9: We will have to deploy the runtime again as the chaincode has changed. Browse into the ‘script’ folder

```
cd script
```

Step 10: Install the new chaincode by running the following script

```
./updateRuntime.sh
```

Step 11: To validate if Event Subscription works, open terminal in visual studio code and cd into following directory

```
cd client_app/test_controller/
```

Step 12: Run the test script using nodejs as below;

We will call issueCertificate() transaction so that the event is emitted

```
node step_x.js CertificateID-1523948684198
```

The result success with details of certificate and monitor method called signifies that our subscription was successful

```
Mustafas-MBP:test_controller mustafahusain$ node step_x.js
...... subscribeForEvents .......
event registration complete
...... issueCertificate .......
student_id: Student-1523948446529
John Doe
MBA
{ result: 'Success' }
_monitor _event received: certificateIssuedEvent
```

Task 3 is complete!

Test Your Knowledge

What class do you need to connect to Business Network?
- BusinessNetworkDefinition
- BusinessNetworkConnection
- FactoryClass
- BusinessNetworkCardStore

In which file do you define static queries?
- .ACL file
- .CTO file
- .QRY file
- .SH file

Client code cannot create an instance of the Admin & Business Network Connection in the same code
- True
- False

Execution of the Named query using the API require you to use the *businessNetworkConnection.buildQuery()* function:
- True
- False

A subscriber application using the composer API receives the event data in the form of
- JSON
- Array
- Message
- UTF8 String

CHAPTER 7: CREATING FRONT END INTERACTIVE INTERFACES

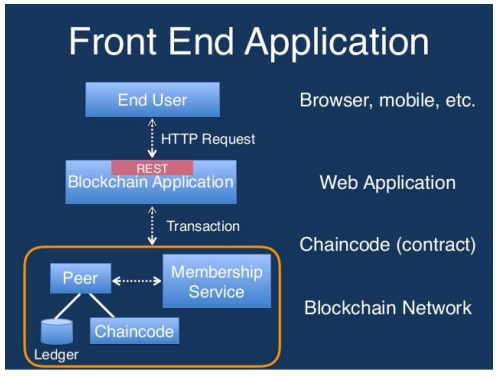
Theory

This chapter again will be completely hands-on. In this chapter we will build a complete end-to-end interactive Use Case utilizing the University example that we have built so far. We will cover all the missing links and FAQs

- How do we invoke the chaincode from an HTML or a frontend application?
- How do we setup the node server and routes to run the client app?
- How does the bloackchain look like in a real work scenario?
- How can a end-user access the blockchain?

Front End Application Patterns

1. Composer Rest Server middleware Architecture:

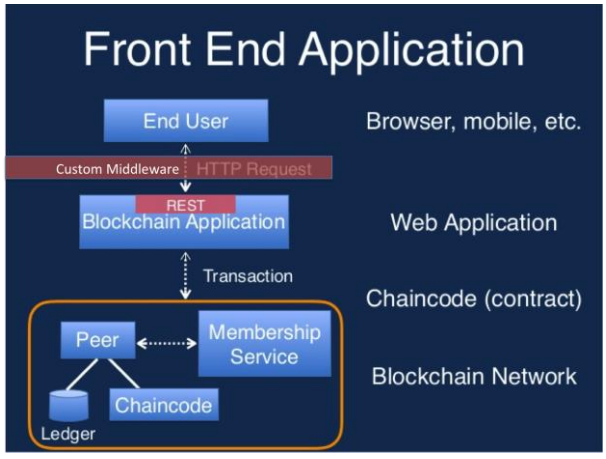


Considerations:

- Rest server must be secured – HTTPS
- Should use authentication – [Passport]
- Should use multi-user mode for rest api

2. Custom middleware pattern:

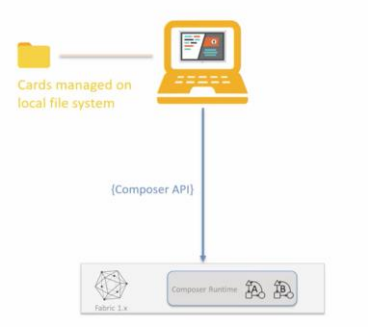
This is the pattern we will be using in our Demo Use Case.



Advantages:

- More secure
- Better control and can plugin existing enterprise applications

3. Desktop Application Architecture



Pros:

- Most Secure

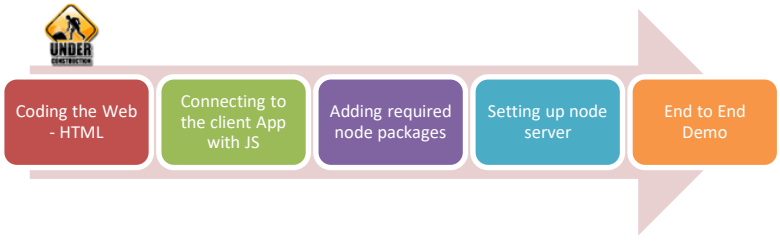
Cons:

- App distribution

Lab 5: Building Interactive Frontend

Tasks

1. Coding the Web – HTML
    1. Directory Structure CSS and Script includes
    2. Coding HTML for different sections:
        i. University
        ii. College
        iii. Student
        iv. Verifier
        v. Historian
    3. Other Model Dialogs
2. Connecting to the client App with JS
    1. On load functions
    2. Javascript to connect HTML to client App
3. Adding required node packages
4. Setting up node server
5. End to End Demo



TASK#1: Coding the Web – HTML

Step 1: Open 'Chapter 5' code provided in Visual Studio Code.

*#**Info**: 'Chapter 5' is a Business Network Scaffolding as in Chapter#4.*

Additionally its has the following which we will build as a step by step process

1. *'html' folder which contains all the frontend code*
2. *Updated 'index.js' file*
3. *Updated 'pacakage.json' file*
4. *'router.js' added in 'client_app/controller' folder*

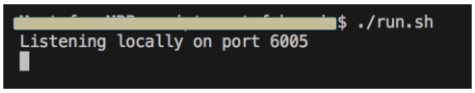
Step 2: Open terminal window of Visual Studio code and type run the following command

```
npm install
```

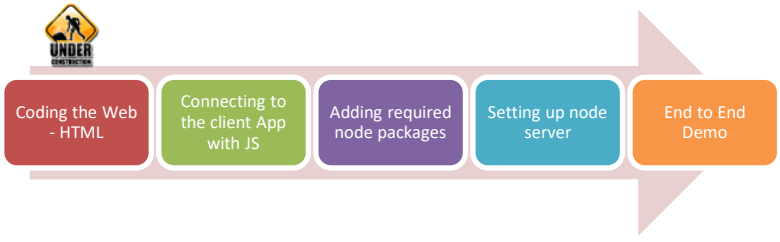
Step 3: Cd into the 'script' folder and run 'run.sh' script to start the node server so that we can launch the frontend for a preview

```
cd script
./run.sh
```

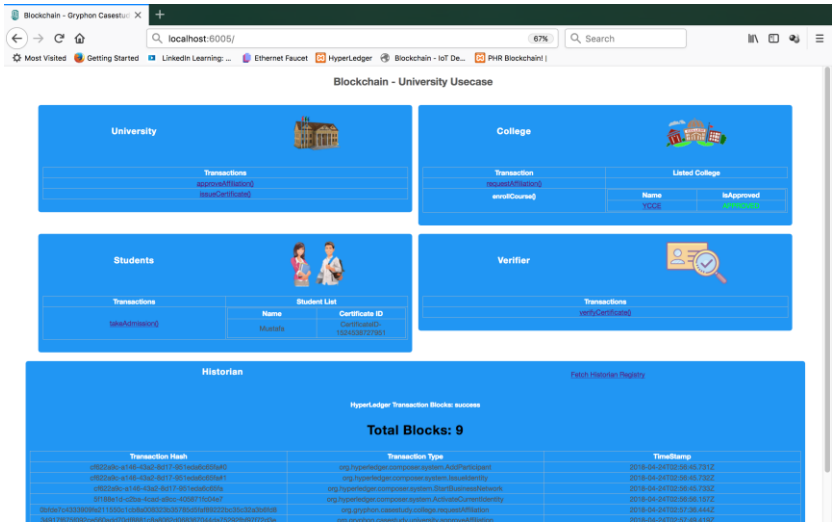
The node server will startup and show the port + address where you can view the frontend html pages in the browser.



Step 4: Launch internet browser and type the URL as below with port number as displayed in previous step:



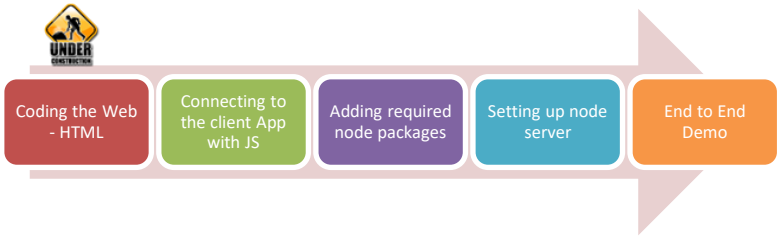
URL: localhost:6005



The UI has five sections, which we will build one by one:

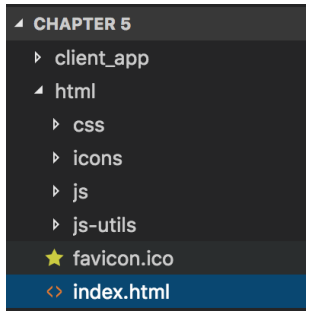
- University Section with Two clickable links
- College Section with clickable actions and table to display college list
- Student Section with clickable actions and table to display student list
- Verifier Section with a clickable link (Transaction) action
- Historian Section that displays all the Blocks/Transactions that have occurred so far

Let's build it step by step…



TASK#1.1: Directory Structure CSS and Script includes

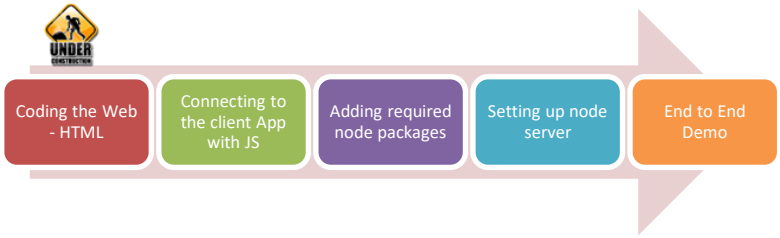
Step 1: Browse to directory 'html'



The 'html' folder has following contenst:

- Css folder:
  - Bootstrap CSS
  - Fonts CSS
  - Application CSS
- Icons:
  - Icons / Images used in our App Use Case
- Js:
  - Application JavaScript file: 'event.js'
- Js-utils:
  - JavaScript files for the 3rd party libraries like jquery and bootstrap
- Favicon.ico: Default icon for the browser (near url)
- Index.html: Default and primary html file for the frontend

Step 2: Under 'html' folder open the file 'index.html'



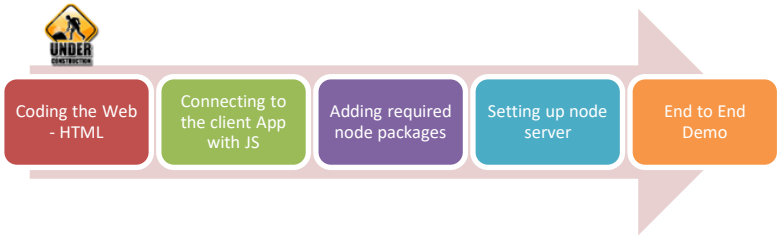
Step 2: Under '<head>' section of the file 'index.html', lets include the css files we need for our UI

```
<head>
    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Blockchain - Gryphon Casestudy University</title>
    <!-- build:css styles/main.css -->
    <link rel="stylesheet" href="css/main.css">
    <link rel="stylesheet" href="css/font-awesome.min.css"  type="text/css">
    <!-- endbuild -->
</head>
```

Step 3: Go to the End of the file 'index.html', and lets include the js files from both 'js' and 'js-util' folder

```
    <!-- build:js scripts/main.js -->
    <script src="js-utils/jquery-3.1.0.min.js"></script>
    <script src="js-utils/jquery-ui.js"></script>
    <script src="js-utils/bootstrap.min.js"></script>
    <script src="js/events.js"></script>
    <!-- endbuild -->

</body>
</html>
```

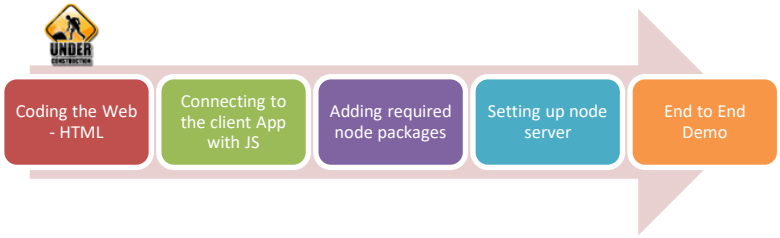
Task 1.1 is complete!



TASK#1.2: Coding HTML for different sections

Step 1: Browse to directory ‘html’ and open file ‘index.html’. Look and code the university section


```
<!-- University Section -->
<div class="sidekick">
    <div>
        <table> <!-- Title & The University Logo -->
            <tr>
                <td> <h1>University</h1> </td>
                <td> <img src="./icons/uni.png"/> </td>
            </tr>
        </table>
    </div>
    <span/>
    <table id="tt"> <!-- Display Transactions that can be done by University -->
        <tr>
            <th>Transactions</th>
        </tr>
        <tr>
            <td> <a class="js-open-modal" href="#" data-modal-id="approveAffiliation">
                approveAffiliation()</a> </td>
        </tr>
        <tr>
            <!-- Will Call Javascript Method "generateCertificate()" when link clicked -->
            <td> <a href="#" onClick="generateCertificate()">issueCertificate()</a> </td>
        </tr>
    </table>
</div>
```

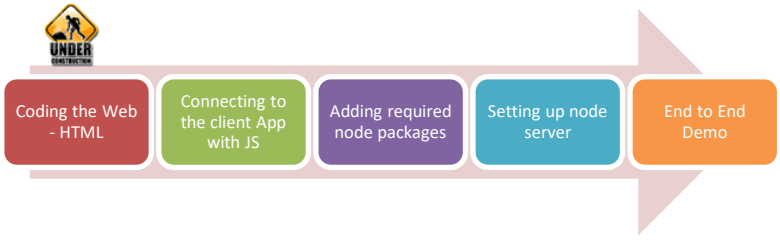


Step 2: Coding College Section

```
<!-- College Section -->
<div class="sidekick">
    <table>
        <tr> <!-- Title & The College Logo -->
            <td> <h1>College</h1> </td>
            <td> <img src="./icons/college.png"/> </td>
        </tr>
    </table>
    <span/>
    <!--div id='invoke_details'></div-->
    <div id="hash">
        <table id="tt"> <!-- Display Transactions that can be done by College -->
            <tr>
                <th>Transaction</th>
                <th>Listed College</th>
            </tr>
            <tr>
                <td> <a class="js-open-modal" href="#"
                    data-modal-id="requestAffiliation">requestAffiliation()</a> </td>
            </tr>
            <tr>
                <td> <h4>enrollCourse()</h4> </td>
                <td> <div id="college_list"></div> </td>
            </tr>
        </table>
    </div>
</div>
```

ID for Modal Dialog that will pop up onclick

College List will be dynamically generated & replaced in this division block

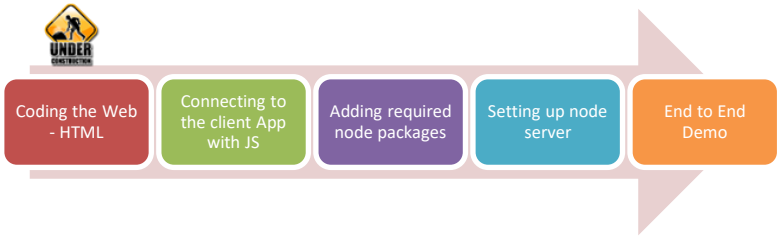


Step 3: Coding Student Section (Similar to College section with changes in Transaction function, images and corresponding texts)

```
<!-- Studen Section -->
<div class="sidekick">
    <table>  <!-- Title & The  Logo -->
        <tr>
            <td> <h1>Students</h1> </td>
            <td> <img src="./icons/stud.png"/> </td>
        </tr>
    </table>
    <span/>
    <table id="tt"> <!-- Display Transactions that can be done by Student -->
        <tr>
            <th>Transactions</th>
            <th>Student List</th>
        </tr>
        <tr>
            <td> <a href="#" onClick="takeAdmission()">takeAdmission()</a> </td>
            <td> <div id="student_list"></div> </td>
        </tr>
    </table>
</div>
```

Step 4: Similarly code the verifier Section

```
<!-- Verifier Section -->
<div class="sidekick">
    <table>
        <tr>
            <th> <h1>Verifier</h1> </th>
            <th> <img src="./icons/Verifier.png"/> </th>
        </tr>
    </table>
    <span/>
    <div id="hash">
        <table id="tt">
            <tr> <td>Transactions</td> </tr>
            <tr>
                <td> <a class="js-open-modal" href="#"
                    data-modal-id="verifyCertificate">verifyCertificate()</a><br/> </td>
            </tr>
        </table>
    </div>
</div>
<!-- Verifier Section End -->
```



Step 5: Coding the Last section i.e Historian that has a link that invokes the javascript and returns the complete history table and that is displayed in the div section with name **id="historian"**

```
<div class="sidekick" style="width:95%">
    <div>
        <table>
            <tr>
                <td> <h1>Historian</h1> </td>
                <td> <a href="#" onclick="getHistorian()">Fetch Historian Registry</a> </td>
            </tr>
        </table>
    </div>
    <span/>
    <div id="historian"></div>
</div>
```

Step 6: Coding Modal Dialogs

```
<!-- Modal -->
<div id="requestAffiliation" class="modal-box">
    <header>
        <a href="#" class="js-modal-close close">x</a>
        <h3>Request Affiliation</h3>
    </header>
    <div class="modal-body">
        <input type="text" id="college_name" placeholder="College Name"></input>
        <div id="college_progress"></div>
    </div>
    <footer>
        <a href="#" class="js-modal-btn" onclick="requestAffiliation()">Submit</a>
        <a href="#" class="js-modal-close">Close</a>
    </footer>
</div>
```

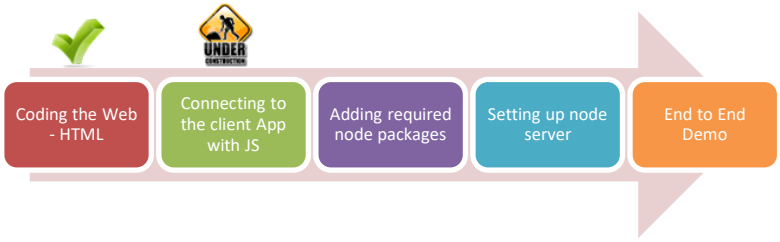
ID for Modal Dialog that will pop up onclick

Will be replaced by spinner image to show progress

Submit the Transaction

Step 7: Similarly coding for other Modal Dialogs

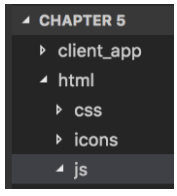
Task 1 is complete!



TASK#2: Connecting the Frontend to call Client App

Step 1: Open ‘Chapter 5’ code provided in Visual Studio Code. Browse to the directory ‘html/js’ and open file ‘event.js’

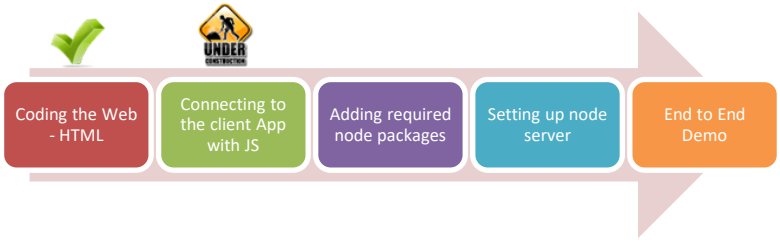
Step 2: Let’s code the methods that we have declared with ‘OnClick’ in HTML file in Task#1.



Step 2: Coding requestAffiliation() method that invokes the Client App rest api as below;

```
function requestAffiliation() {
    // Show Progress until the task is complete
    document.getElementById('college_progress').innerHTML = '<img src="./icons/progress.gif"/>';

    // Get the attributes from the UI i.e College Name
    let college_name = document.getElementById('college_name').value;

    // Set the Payload for the Post restapi call
    let options = { 'college_name': college_name };
    {
        // Use Async ajax call to post a request to the Client App
        $.when($.post('/composer/client/requestAffiliation', options)).done(function (_res) {

            // End progress display
            document.getElementById('college_progress').innerHTML = '';

            // Update the UI with college list
            displayCollegeList();

            // Close the Modal Dialog
            closeModal();
        });
    }
}
```

Will will setup router to call client app api in next Task

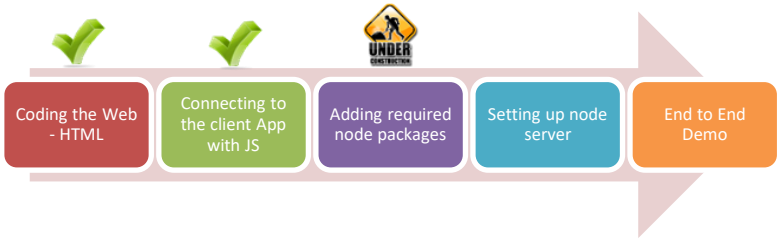


Step 3: Retrieving list of colleges from the client App and processing it to be displayed in HTML

```
/**
 * Display Colleges
 */
function displayCollegeList() {

    // Use Async ajax call to get a list of colleges from the Client App
    $.when($.get('composer/client/getCollegeList')).done(function (_res) {

        // Successful response received, lets check it on browser console
        console.log(_res.college_list);

        let _str = '';
        let _nstr = '';

        // We will now build the Table to be displayed in UI
        _str += '<table><tr><th>Name</th><th>isApproved</th></tr>';
        _res.college_list.forEach(function (_row) {
            // Check is college is approved or NOT aaproved by university
            let td = (_row.is_approved == 0) ? '<td class="red"> NOT APPROVED</td>'
                    : '<td class="green"> APPROVED</td>';

            _str += '<tr><td>' + '<a href="#" onClick=enrollProgram("' + _row.id + '")>'
                    + _row.name + '</a></td>' + td + '</tr>';
            if (_row.is_approved == 0) {
                _nstr += '<input type="checkbox" name="collegeIds" value="'
                    + _row.id + '">' + _row.name + '</input></br>';
            }
        })
        _str += '</table>';

        // Display College List
        document.getElementById('college_list').innerHTML = _str;
        document.getElementById('approve_list').innerHTML = _nstr;
    });
```

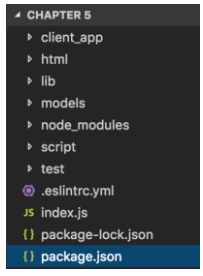
Step 4: Similarly we can code other methods to call the client App methods and process them to be displayed in HTML. Please visit ‘events.js’ for other methods.

Task 2 is complete!

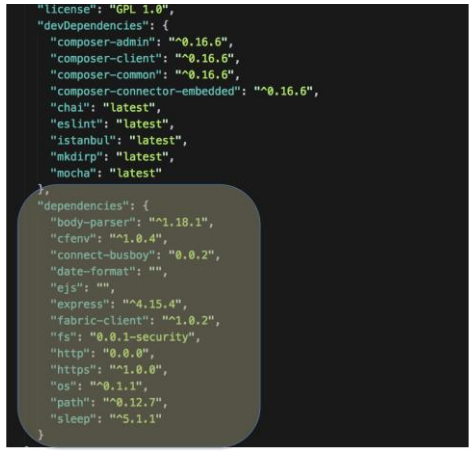


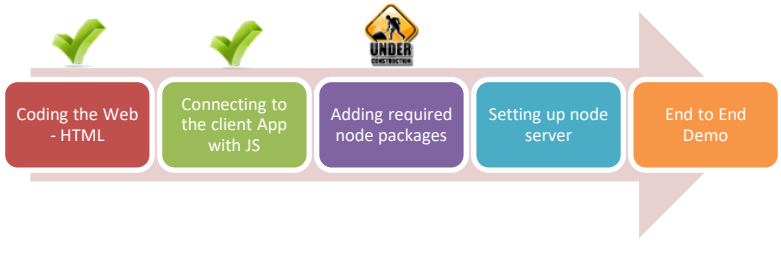
TASK#3: Adding dependency packages

Step 1: Open ‘Chapter 5’ code provided in Visual Studio Code. Browse to the open file ‘package.json’



Step 2: Let’s add dependency packages that we will need for our nodejs server and routing the restapi calls

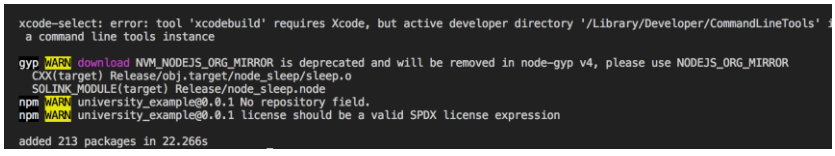




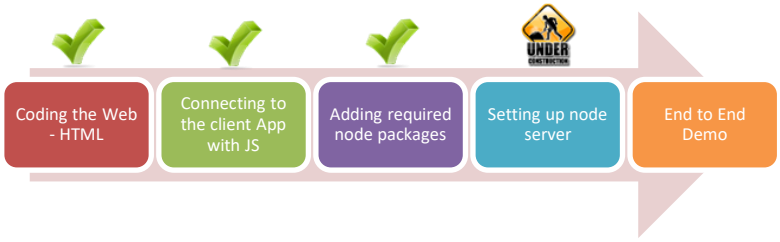
Step 3: Open terminal window of Visual Studio code and type run the following command

```
npm install
```

There might be some warnings but that is normal. Packages are now added to the system

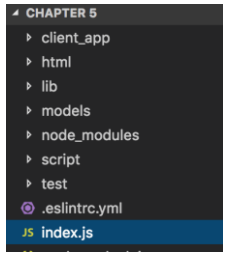


Task 3 is complete!

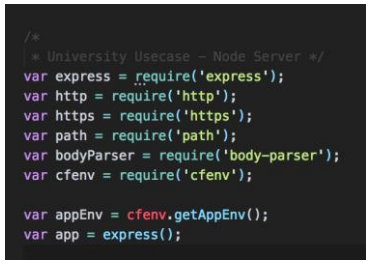


TASK#4: Setting up the node server

Step 1: Open ‘Chapter 5’ code provided in Visual Studio Code. Browse to the open file ‘index.js’ in the main folder

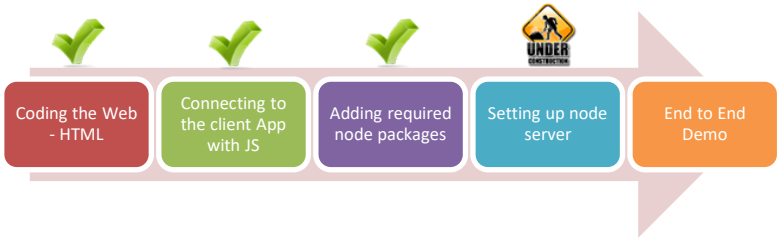


Step 2: Open ‘Chapter 5’ code provided in Visual Studio Code. Browse to the open file ‘index.js’ in the main folder



```
/*
 * University Usecase - Node Server */
var express = require('express');
var http = require('http');
var https = require('https');
var path = require('path');
var bodyParser = require('body-parser');
var cfenv = require('cfenv');

var appEnv = cfenv.getAppEnv();
var app = express();
```

- Define required packages
- Setup the environment variable for selecting the right port



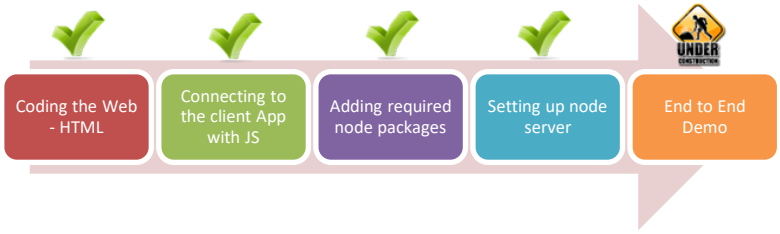
Step 3: Define where are the views, json parsing and application name for port registration

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.set('appName', 'org.gryphon.casestudy');
app.set('port', appEnv.port);

app.set('views', path.join(__dirname + '/html'));
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/html'));
app.use(bodyParser.json());
```

Step 4: Define restapi router and start listening to the port for rest api calls

```
// Define your own router file in controller folder, export the router, add it into the index.js.
// app.use('/', require("./controller/yourOwnRouter"));
app.use('/', require("./client_app/controller/router"));

if (cfenv.getAppEnv().isLocal == true)
  {
    var server = app.listen(app.get('port'), function() {
    console.log('Listening locally on port %d', server.address().port);}); }
  else {
    var server = app.listen(app.get('port'), function() {
      console.log('Listening remotely on port %d', server.address().port);});
  }
```

Task 4 is complete!



TASK#5: Running the use-case End to End

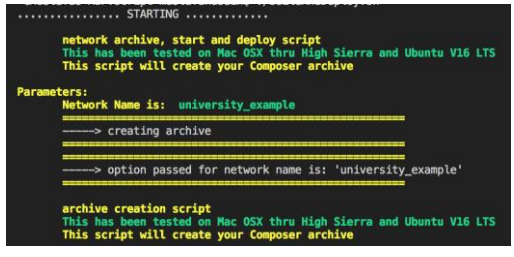
Step 1: Open 'Chapter 5' code provided in Visual Studio Code. Open the terminal window and cd into the scripts folder

```
cd script
```

Step 2: Let's completely build and deploy the blockchain application. Use the following script

```
./buildAndDeploy.sh
```

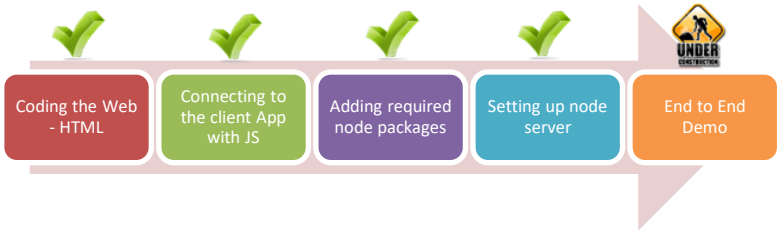
This will start the Hyperledger Fabric, install the Business Network Application and created the Peer / Admin cards

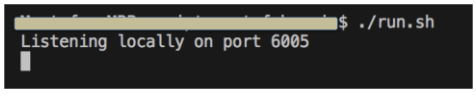


Step 3: Now let's run the rest server using the following script

```
./run.sh
```

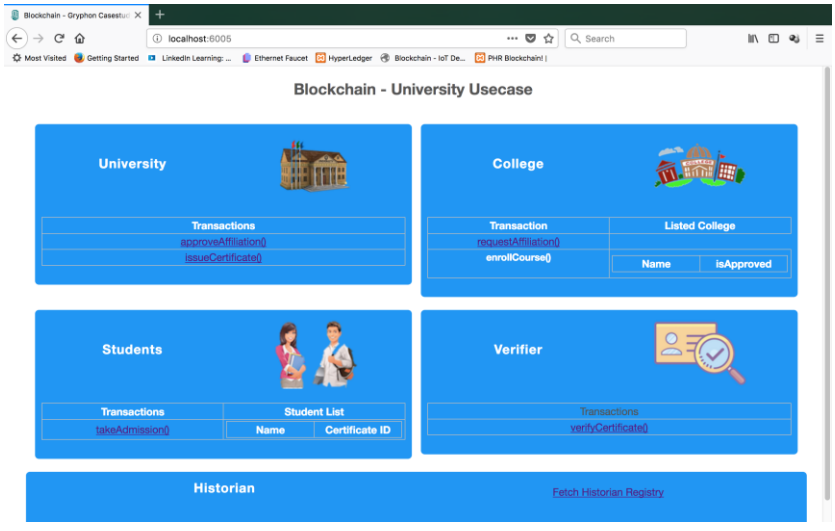
The node server will startup and show the port + address where you can view the frontend html pages in the browser.





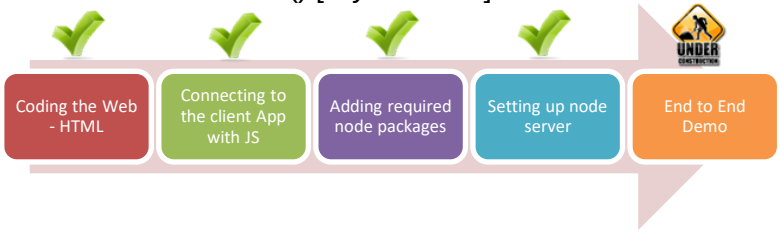
Step 4: Launch Internet browser and type the URL as below with port number as displayed in previous step:

URL: localhost:6005 *(Use Port displayed on your screen)*



Step 5: Our use-case UI is ready for action, the sequence of order of transactions will be:

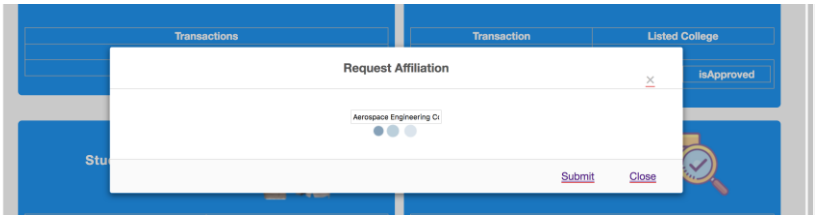
1. requestAffiliation() [By College]
2. approveAffliation() [By University]
3. enrollProgram() [By College]
4. takeAdmission() [By Student]



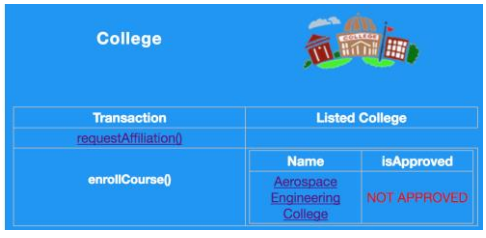
5. issueCertificate() [By University]
6. verifyCertificate() [By Verifier]
7. Finally we will check the transaction history using the Historian

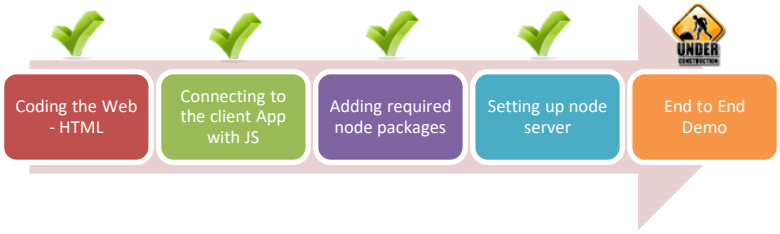
Let's execute the first transaction by clicking requestAffiliation() in College section.

Step 6: In the popup dialog enter any college name and click submit:



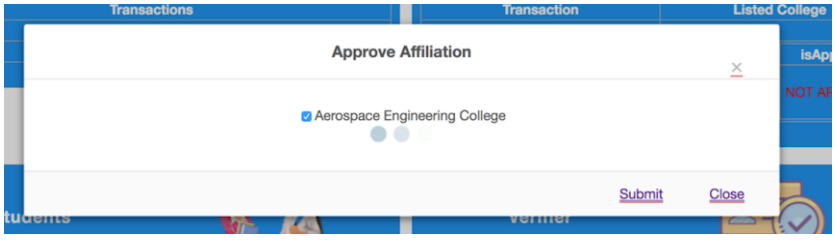
Progress image will show up and when transaction is complete, college will be listed in the College Section:



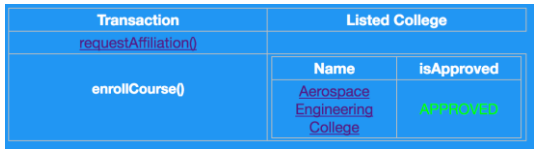


Step 7: Now from University Panel click on the approveAffiliation() link

Step 8: On the new popup dialog, select the college checkbox and click submit

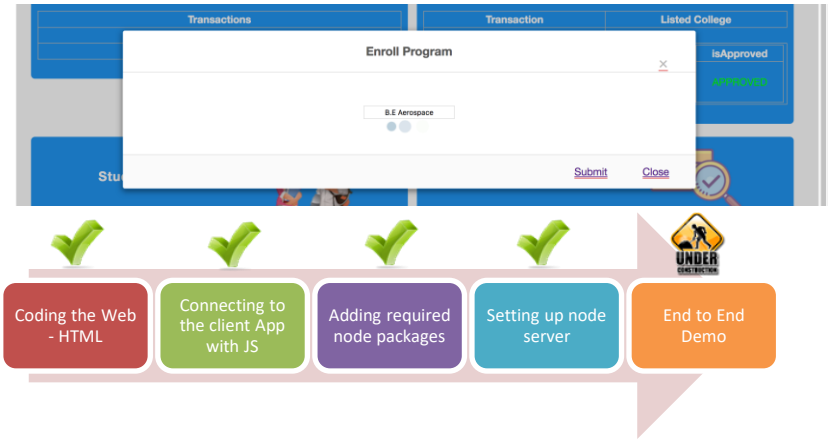


The college status should show up in green and as 'Approved'



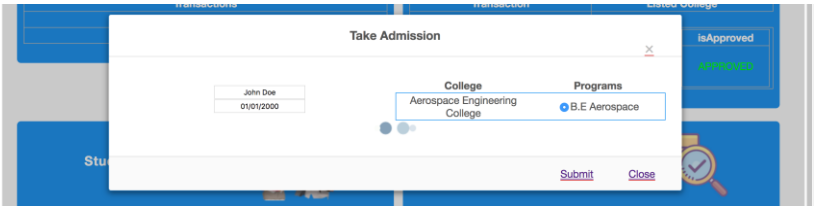
Step 9: Now college can start enrolling new programs. Click the college name to enroll program.

Step 10: A popup dialog asking for Program name should open up. Enter any name of the program and click submit



Step 11: Students can now take admission to the college for the respective programs offered by the college. Under Student section, click on takeAdmission() transaction link

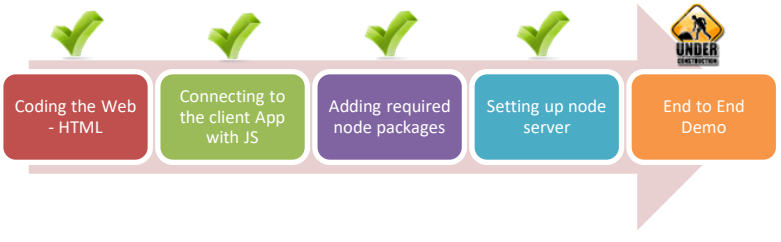
Step 12: On the new opened dialog, enter the Student Name, date of birth and select appropriate college and program. Then click submit.

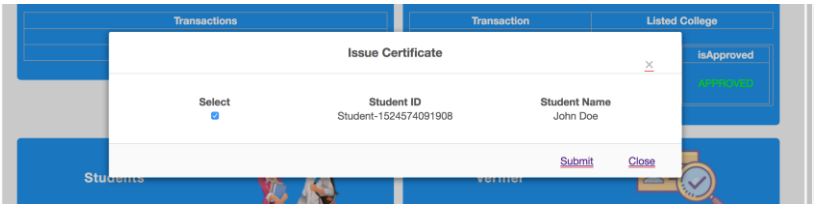


Student should be enrolled and student's name starts showing up on the Student section

Step 13: Student has now completed the program and University will issue the certificate. Click on issueCertificate() under University section to generate the certificate for the student

Step 14: Select the student who has to be awarded the certificate and click submit



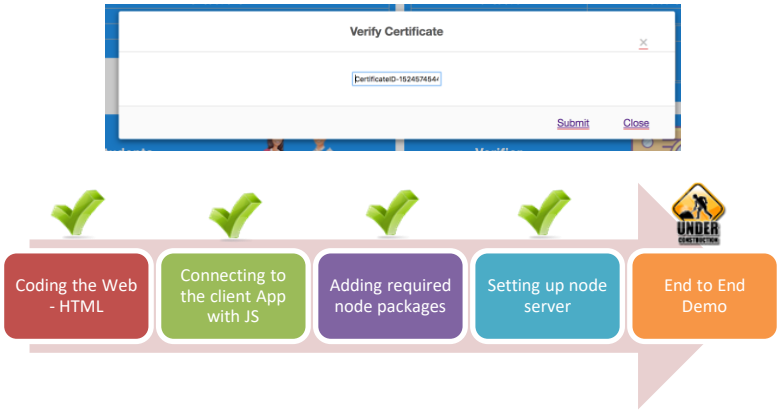


Step 14: Select the student who has to be awarded the certificate and click submit

Transactions	Student List	
	Name	Certificate ID
takeAdmission()	John Doe	CertificateID-1524574544346

A certificate is generated and the Certificate unique ID is displayed against the student.

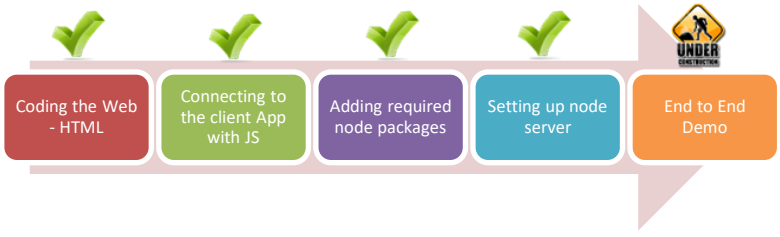
Step 15: Select the CertifcateID and copy it in clipboard. Now for verification click on verifyCertificate() under the verifier section.



Step 16: Enter the generated certificate ID on the new window and click submit.

Certificate of completion of the program will be displayed. The Asset is now in blockchain, secured and uniquely identifies the student's authentic academics.

Step 17: Blockchain represent an un-mutable set of transactions. To check what transactions have occurred on the blockchain and validate the authenticity of the transaction that created the certificate we can view the Historian – i.e the transaction log.



Step 18: To view the Historian, click on the 'Fetch Historian Registry' link in the bottom section.

Complete list of transaction with their unique ID and time are displayed.

Task 5 is complete!

SUMMARY

End to End use-case for Hyperledger is now ready for demo! Students are encouraged to use and exercise the UI and the functionality defined here and better it.

Please be aware that this is just for learning and many corner test-cases have not been considered.

Further to this there are few advanced topics that we shall cover in our next chapters like:

- Securing the Rest Server
- Adding Organizations & Creating the channels

In this chapter we have learned the following:

- Developing Interactive frontend for Hyperledger
- Bridging the UI with client App using javascript calls
- Setting up node server and routing rest-api calls to call the client app
- Invoking Client App API

Supplemental Lab Projects Using Hyperledger 1.1

LAB: Deploying HyperLedger 1.1 Fabric Business Network

Prerequisites: Hyperledger Fabric rev. 1.2.

Objective:

This lab describes creating a business network and deploying chaincode using the Hyperledger Fabric v1.2.0-preview code base (found here: https://github.com/hyperledger/fabric.git). This lab exploits the Hyperledger Fabric v1.2.0-preview configuration toolset to create the business network consisting of the following items:

- A single orderer
- Three peers (each part of separate organization)
- A single channel
- Example chaincode

At the end of this lab, you will have constructed a running instance of Hyperledger Fabric business network, as well as deployed, instantiated, and executed chaincode. In addition, our network will have TLS enabled.

The lab describes the context of fictional "bta.com" company and its three organizations: Org1, Org2 and Org2.

For our lab we will make use of the Fabric deployment model based on Docker containers. Our network will run on Docker containers running on our target localhost.

Generate Peer and Orderer Certificates

Nodes (such as peers and orderers) are permitted to access business networks using a membership service provider, which is typically in the form of a certificate authority. In this example, we use the development tool named cryptogen to generate the required certificates. We use a local MSP to store the certs, which are essentially a local directory structure, for each peer and orderer. In production environments, you can exploit the fabric ca toolset introducing full-featured certificate authorities to generate the certificates. In addition, our network implements TLS (transport layer security) when authenticating remote procedure calls (using grpc protocol for communication).

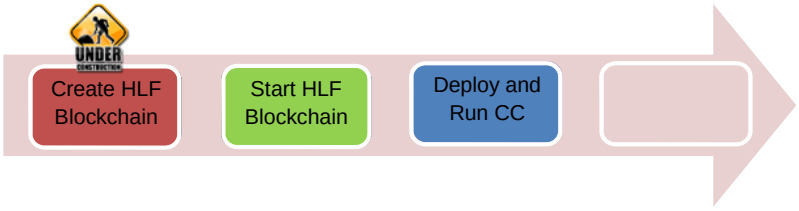
A cryptogen tool uses a yaml configuration file as its configuration - based on the content of this file, the required certificates are generated. We are going to use crypto-config.yaml file from the fabric-samples\first-network directory as a

base for our configuration. We are going to define three organizations for peers and single orderer organization.

Here is the listing of our crypto-config.yaml configuration file (for simplicity, all comments are removed from this listing):

```
OrdererOrgs:

  - Name: Orderer
    Domain: bta.com
    Specs:
      - Hostname: orderer
PeerOrgs:
  - Name: Org1
    Domain: org1.bta.com
    Template:
      Count: 1
    Users:
      Count: 1
  - Name: Org2
    Domain: org2.bta.com
    Template:
      Count: 1
    Users:
      Count: 1
  - Name: Org3
    Domain: org3.bta.com
    Template:
      Count: 1
    Users:
      Count: 1
```

Lab: Create a Business Network



Task 1: Create HLF Blockchain

LAB: Creating a Hyperledger Fabric Business Network

Step 1: To generate certificates, run the following commands:

bash

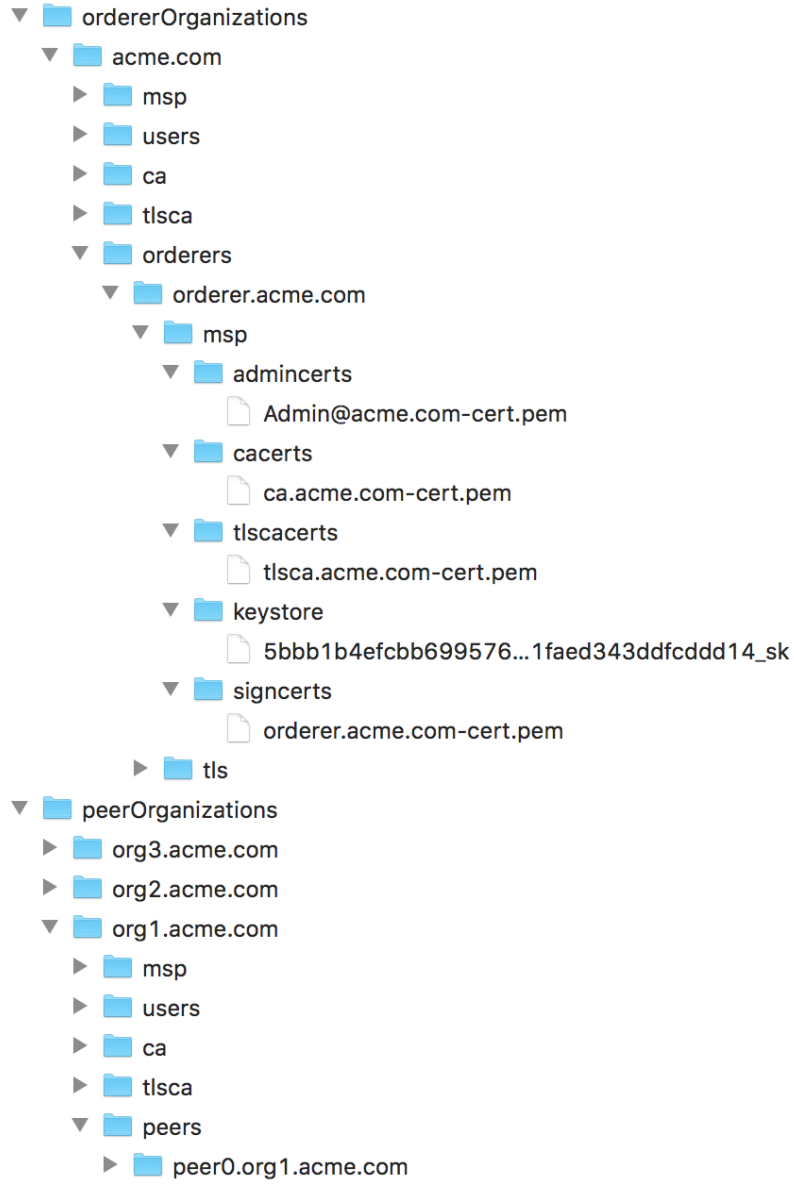
cd /home/fenago/fabric-samples/basic-network/

cryptogen generate --config=./crypto-config.yaml

After running of cryptogen tool you should see the following output in console:

```
[root@linux391 basic-network]# cryptogen generate --config=./cry
org1.acme.com
org2.acme.com
org3.acme.com
[root@linux391 basic-network]#
```

In addition, the new crypto-config directory has been created and contains various certificates and keys for orderer and peers. See the following screenshot that illustrate the content of this directory.


ordererOrganizations
acme.com
msp
users
ca
tlsca
orderers
orderer.acme.com
msp
admincerts
Admin@acme.com-cert.pem
cacerts
ca.acme.com-cert.pem
tlscacerts
tlsca.acme.com-cert.pem
keystore
5bbb1b4efcbb699576...1faed343ddfcddd14_sk
signcerts
orderer.acme.com-cert.pem
tls
peerOrganizations
org3.acme.com
org2.acme.com
org1.acme.com
msp
users
ca
tlsca
peers
peer0.org1.acme.com

Note: The crypto directories become the local MSP for each of the peers and orderers. This fact becomes clear when we populate the configuration and docker-compose yaml files.

LAB: Introduction to Chaincode 1.2

Prerequisites

- Ensure that the environment pre-requisites are installed: http://hyperledger-fabric.readthedocs.io/en/release-1.2/prereqs.html
- Ensure that the HyperLedger Fabric Samples are installed for version 1.2.0 : http://hyperledger-fabric.readthedocs.io/en/release-1.2/samples.html
- Cryptogen, configtxgen, configtxlater, peer, orderer, and fabric-ca-client should be installed and placed in the /bin directory (/home/fenago/bin --- because it was installed to this location in this lab VM). It is important that this is place in the $PATH :

  export PATH=/home/fenago/bin:$PATH

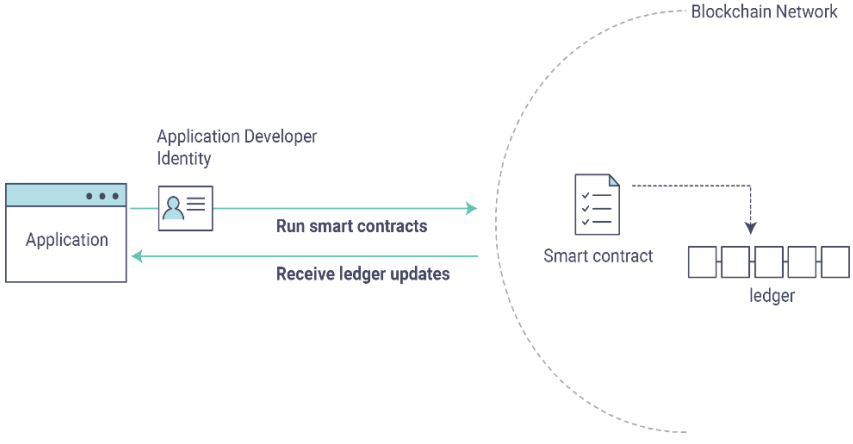
- Update the path in ~./bashrc

Objectives

In this section we'll be looking at a handful of sample programs to see how Fabric apps work. These apps (and the smart contract they use) – collectively known as fabcar – provide a broad demonstration of Fabric functionality. Notably, we will show the process for interacting with a Certificate Authority and generating enrollment certificates, after which we will leverage these generated identities (user objects) to query and update a ledger.

THEORY

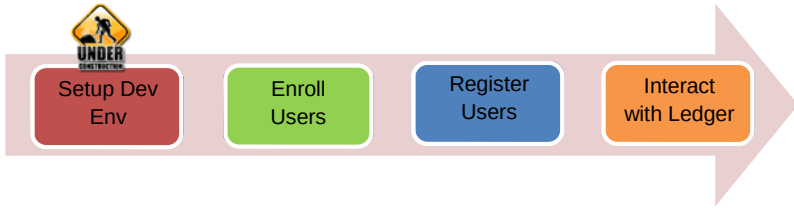
1. Setting up a development environment. Our application needs a network to interact with, so we'll download one stripped down to just the components we need for registration/enrollment, queries and updates:



2. Learning the parameters of the sample smart contract our app will use. Our smart contract contains various functions that allow us to interact with the ledger in different ways. We'll go in and inspect that smart contract to learn about the functions our applications will be using.

3. Developing the applications to be able to query and update assets on the ledger. We'll get into the app code itself (our apps have been written in JavaScript) and manually manipulate the variables to run different kinds of queries and updates.

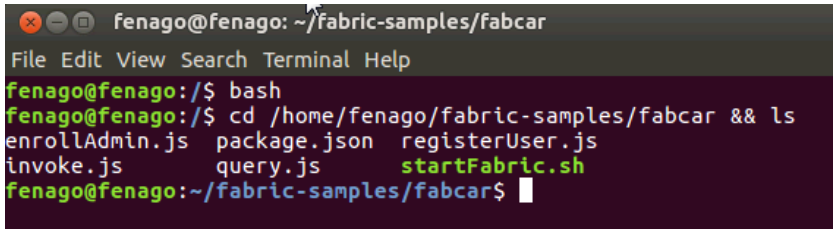
LAB: Setting Up Development Environment



Task 1 : Setup Development Environment

Execute these commands

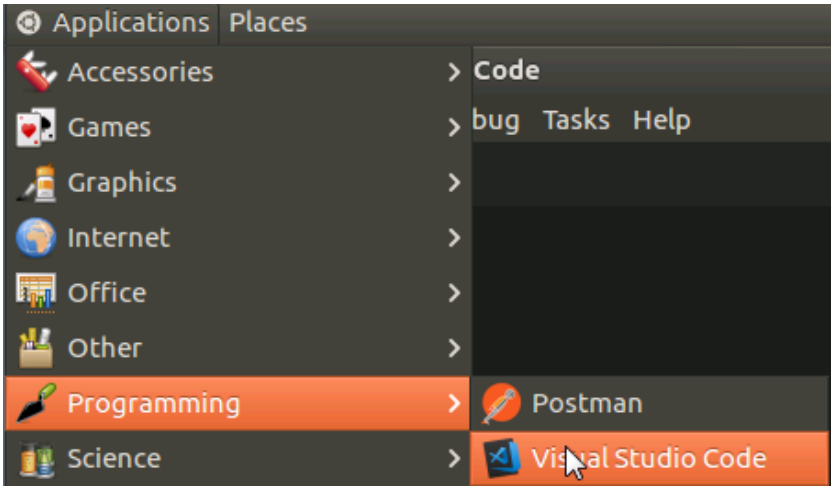
1. bash
2. cd /home/fenago/fabric-samples/fabcar && ls



Clean up existing docker infrastructure by running: docker network prune && docker rmi dev-peer0.org1.example.com-fabcar-1.0-5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba

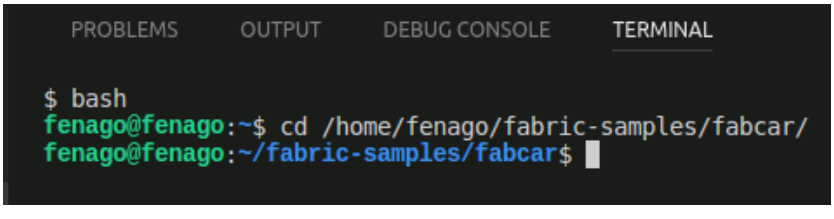
```
fenago@fenago:~/fabric-samples/fabcar$ docker network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
fenago@fenago:~/fabric-samples/fabcar$ docker rmi dev-peer0.org1.example.com-fab
car-1.0-5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba
Untagged: dev-peer0.org1.example.com-fabcar-1.0-5c906e402ed29f20260ae42283216aa7
5549c571e2e380f3615826365d8269ba:latest
Deleted: sha256:f6258e06601786970cba783c7be4f68b19f7cf8aaff8fb51d412fb6c7736c162
Deleted: sha256:15c83f7d08d5861eb8a79cff306366a367001ef899857a3ecffba001d62c5cb2
Deleted: sha256:340de87baa0fa8360cd9598611ac54790b71b2a203c0d67cd261bb004b0062bf
Deleted: sha256:bfebb6471f71d3296d3ec6d1f14d764aebe04b0786cb79ff2feb362ef82130cb
fenago@fenago:~/fabric-samples/fabcar$
```

3. Launch Visual Studio Code (ensure that HyperLedger Composer, Docker, Docker Explorer, Docker Compose, and Go extensions are added. They are already added to the Lab VM)



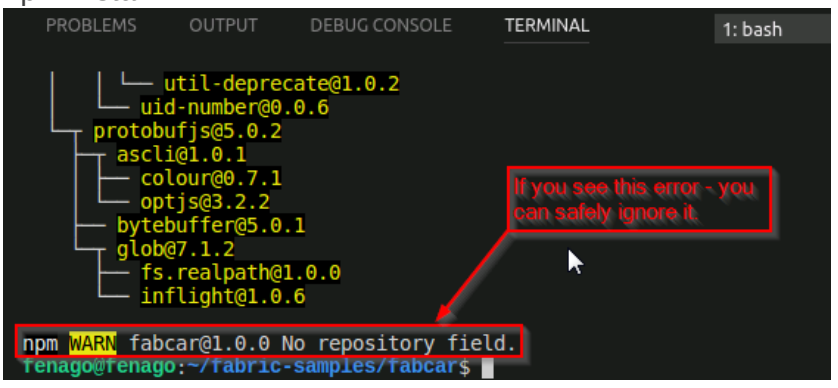
4. In Visual Studio Code, select View → Integrated Terminal.

5. In the Integrated Terminal type:
   bash
   cd /home/fenago/fabric-samples/fabcar/
   //from now on – stay in the Integrated Terminal

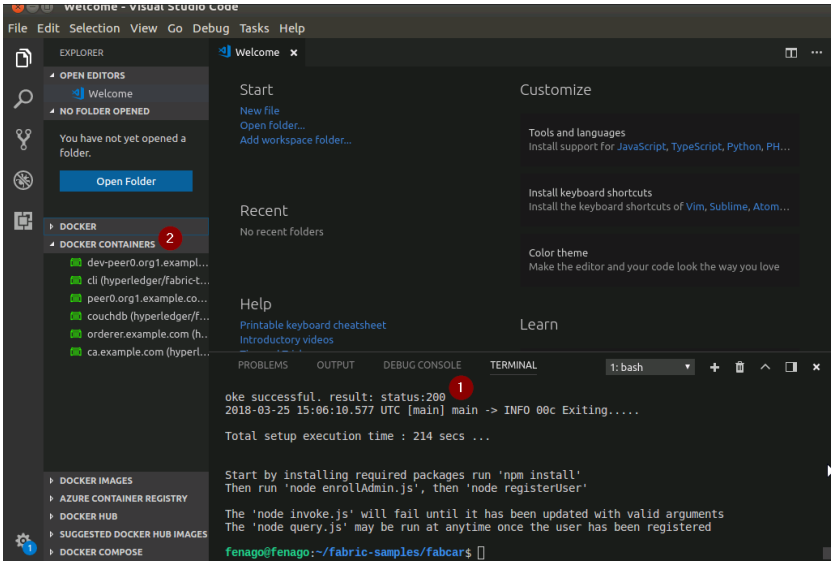


6. Run the following command to install the Fabric dependencies for the applications. We are focused on fabric-ca-client which will allow our app(s) to communicate with the CA server and retrieve identity material, and with fabric-client which allows us to load the identity material and talk to the peers and ordering service.

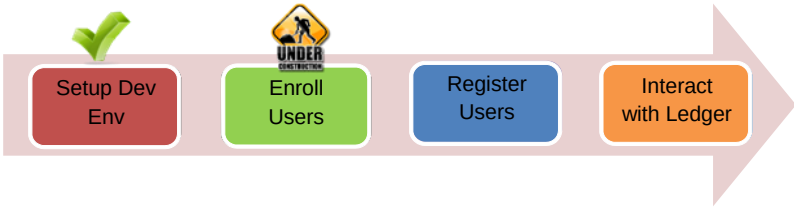
Execute:
npm install



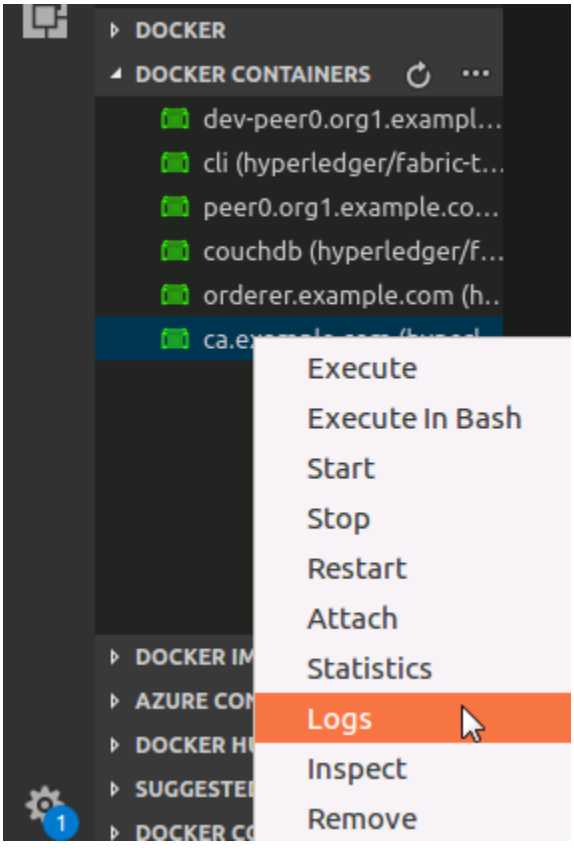
7. Execute
./startFabric.sh
Notice 2 things:
First – it should return a 200 status
Second – notice that Docker Containers have been created. This is HyperLedger Fabric. You can also manipulate them from the command line but VSCode gives you a UI interface to manage Docker.



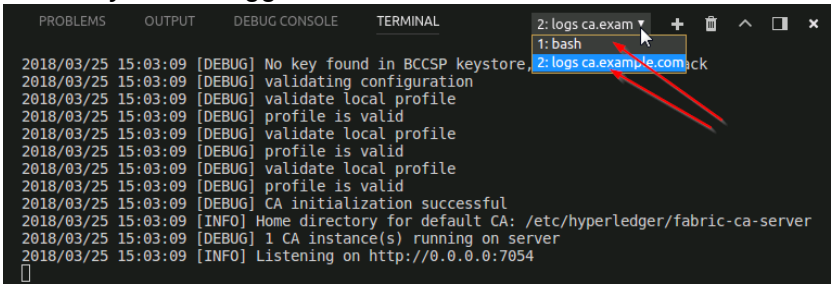
Task 2: Enroll the Admin Users



1. Stream the logs on the CA Server by right clicking on ca.example.com (this will tail 50 -f the server log file).



NOTE: you can toggle between terminals:

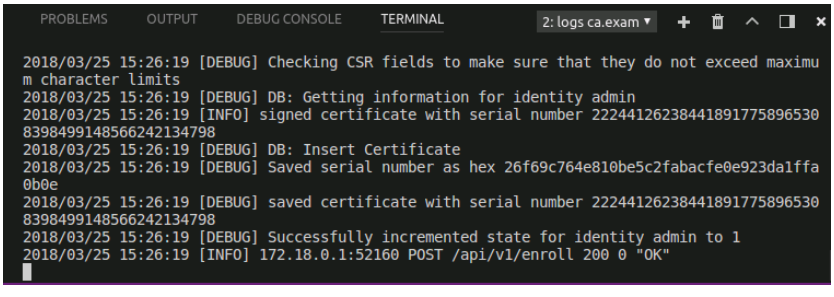


2. When we launched our network, an admin user (admin) was registered with our Certificate Authority. Now we must send an enroll call to the CA server and retrieve the enrollment certificate (eCert) for this user. We won't cover enrollment details here, but the SDK and by extension our applications need this cert in order to form a user object for the admin. We will then use this admin object to subsequently register and enroll a new user.
Execute:
Go back to the original bash shell (not the logs of ca.example.com) and:

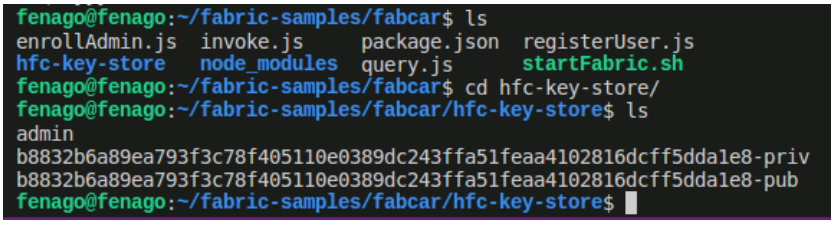
cat enrollAdmin.js
node enrollAdmin.js

you should see this in the bash shell:

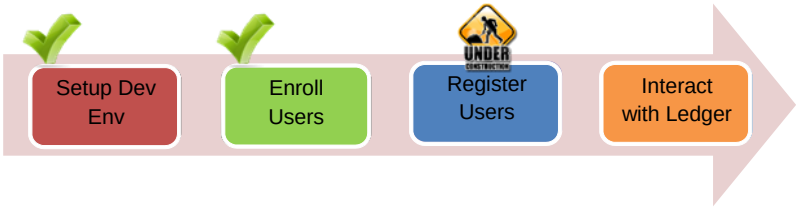
You should see this in the ca.example.com container:



This program will invoke a certificate signing request (CSR) and ultimately output an eCert and key material into a newly created folder - hfc-key-store - at the root of this project. Our apps will then look to this location when they need to create or load the identity objects for our various users.



Register and Enroll Users



1. With our newly generated admin eCert, we will now communicate with the CA server once more to register and enroll a new user. This user - user1 - will be the identity we use when querying and updating the ledger.

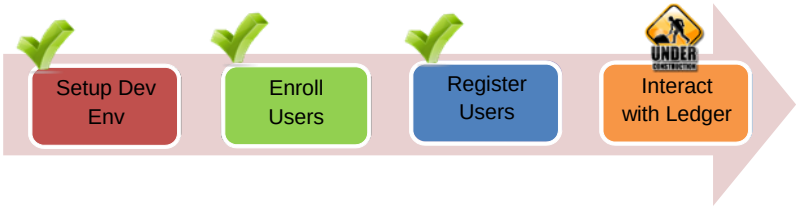
It's important to note here that it is the admin identity that is issuing the registration and enrollment calls for our new user (i.e. this user is acting in the role of a registrar). Send the register and enroll calls for user1:
Execute:
node registerUser.js

```
fenago@fenago:~/fabric-samples/fabcar$ ls
enrollAdmin.js  invoke.js     package.json  registerUser.js
hfc-key-store   node_modules  query.js      startFabric.sh
fenago@fenago:~/fabric-samples/fabcar$ node registerUser.js
 Store path:/home/fenago/fabric-samples/fabcar/hfc-key-store
Successfully loaded admin from persistence
Successfully registered user1 - secret:nVTiVEgeSmgM
Successfully enrolled member user "user1"
User1 was successfully registered and enrolled and is ready to intreact with the fabric net
work
fenago@fenago:~/fabric-samples/fabcar$
```

This will create a CSR and generate the keys and eCerts into the hfc-key-store subdirectory. You should see admin and user1 registered.

```
fenago@fenago:~/fabric-samples/fabcar$ ls
enrollAdmin.js  invoke.js     package.json  registerUser.js
hfc-key-store   node_modules  query.js      startFabric.sh
fenago@fenago:~/fabric-samples/fabcar$ cd hfc-key-store
fenago@fenago:~/fabric-samples/fabcar/hfc-key-store$ ls
046bb0629418a27461abde43b056b443f61c5fcb277b07a3caab5e398e39d88d-priv
046bb0629418a27461abde43b056b443f61c5fcb277b07a3caab5e398e39d88d-pub
admin
b8832b6a89ea793f3c78f405110e0389dc243ffa51feaa4102816dcff5dda1e8-priv
b8832b6a89ea793f3c78f405110e0389dc243ffa51feaa4102816dcff5dda1e8-pub
user1
fenago@fenago:~/fabric-samples/fabcar/hfc-key-store$
```

Creating the UI

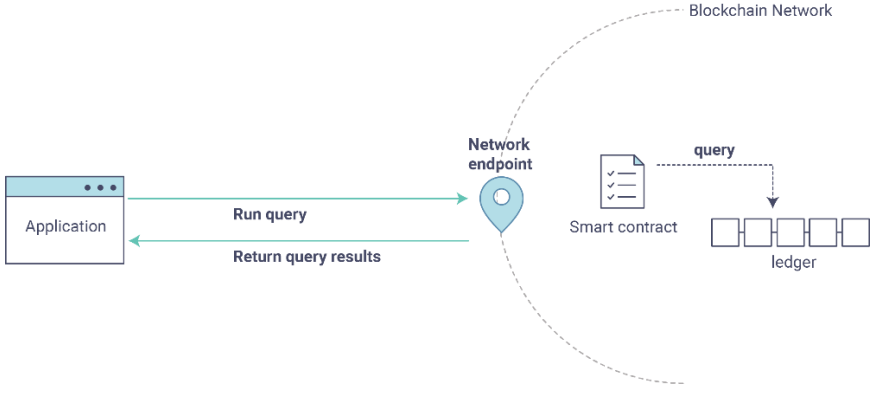


1. Inside of query.js – you will see user1 is set to sign all requests (hence the reason we had to register User1 above)

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   1: view

33          // assign the store to the fabric client
34          fabric client.setStateStore(state store);
35          var crypto suite = Fabric Client.newCryptoSuite();
36          // use the same location for the state store (where the users' certificate
    are kept)
37          // and the crypto store (where the users' keys are kept)
38          var crypto store = Fabric Client.newCryptoKeyStore({path: store path});
39          crypto suite.setCryptoKeyStore(crypto store);
40          fabric client.setCryptoSuite(crypto suite);
41
42          // get the enrolled user from persistence, this user will sign all requests
43          return fabric client.getUserContext('user1', true);
```

Queries are how you read data from the ledger. This data is stored as a series of key/value pairs, and you can query for the value of a single key, multiple keys, or – if the ledger is written in a rich data storage format like JSON – perform complex searches against it (looking for all assets that contain certain keywords, for example).

This is a representation of how a query works:

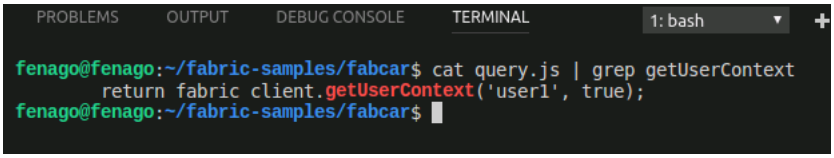


First, let's run our query.js program to return a listing of all the cars on the ledger. We will use our second identity - user1 - as the signing entity for this application. The following line in our program specifies user1 as the signer:

fabric_client.getUserContext('user1', true);

Recall that the user1 enrollment material has already been placed into our hfc-key-storesubdirectory, so we simply need to tell our application to grab that identity. With the user object defined, we can now proceed with reading from the ledger. A function that will query all the cars, queryAllCars, is pre-loaded in the app, so we can simply run the program as is:

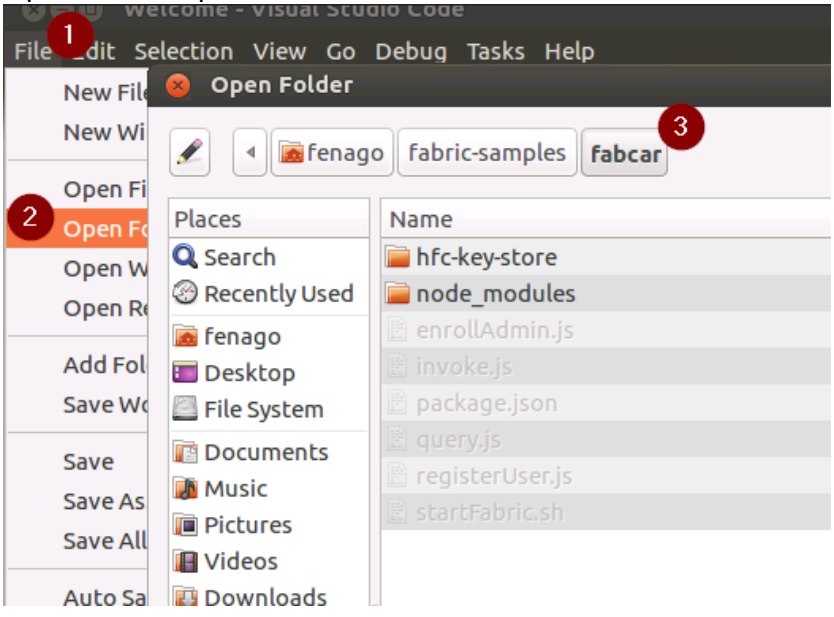
Execute:
cd /home/fenago/fabric-samples/fabcar
cat query.js | grep getUserContext



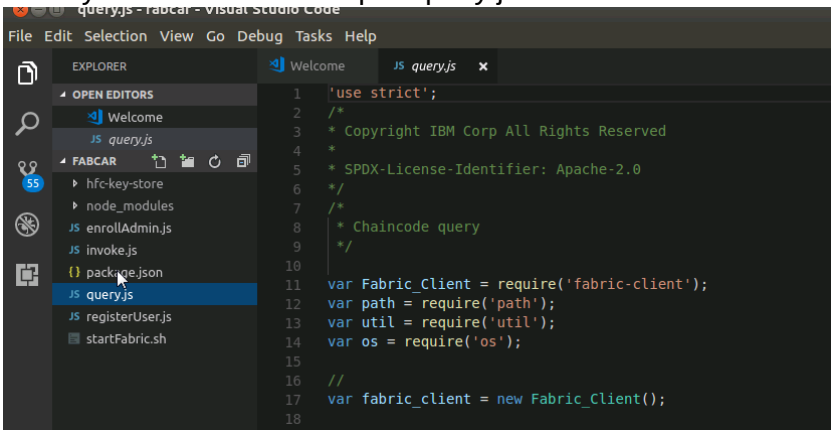
node query.js

These are the 10 cars. A black Tesla Model S owned by Adriana, a red Ford Mustang owned by Brad, a violet Fiat Punto owned by Pari, and so on. The ledger is key/value based and in our implementation the key is CAR0 through CAR9. This will become particularly important in a moment.

2. Open File → Open Folder → Find and select Fabcar



3. Once you click OK – then open query.js in VSCode



The initial section of the application defines certain variables such as channel name, cert store location and network endpoints. In our sample app, these variables have been baked-in, but in a real app these variables would must be specified by the app dev.

var channel = fabric_client.newChannel('mychannel');

var peer = fabric_client.newPeer('grpc://localhost:7051');

channel.addPeer(peer);

var member_user = null;

var store_path = path.join(__dirname, 'hfc-key-store');

console.log('Store path:'+store_path);

var tx_id = null;

This is the chunk where we construct our query:

// queryCar chaincode function - requires 1 argument, ex: args: ['CAR4'],

// queryAllCars chaincode function - requires no arguments , ex: args: [''],

const request = {

//targets : --- letting this default to the peers assigned to the channel

chaincodeId: 'fabcar',

fcn: 'queryAllCars',

args: ['']

};

When the application ran, it invoked the fabcar chaincode on the peer, ran the query AllCarsfunction within it, and passed no arguments to it.

To look at the available functions within our smart contract, navigate to the chaincode/fabcar/go subdirectory at the root of fabric-samples and open fabcar.go in your editor.

Note

These same functions are defined within the Node.js version of the fabcar chaincode.

You'll see that we have the following functions available to call: initLedger, queryCar, queryAllCars, createCar

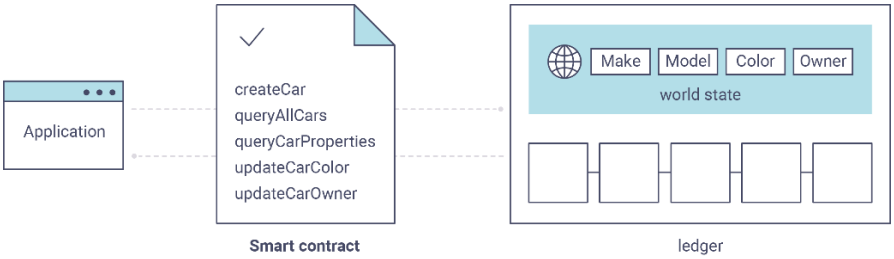
, and changeCarOwner.

Let's take a closer look at the queryAllCars function to see how it interacts with the ledger.

```
func (s *SmartContract) queryAllCars(APIstub shim.ChaincodeStubInterface)
sc.Response {


    startKey := "CAR0"

    endKey := "CAR999"


    resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
```

This defines the range of queryAllCars. Every car between CAR0 and CAR999 – 1,000 cars in all, assuming every key has been tagged properly – will be returned by the query.

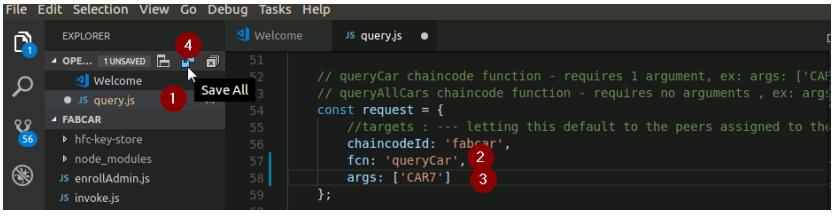
Below is a representation of how an app would call different functions in chaincode. Each function must be coded against an available API in the chaincode shim interface, which in turn allows the smart contract container to properly interface with the peer ledger.



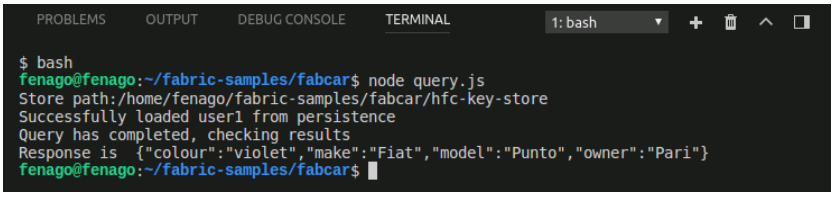
We can see our queryAllCars function, as well as one called createCar, that will allow us to update the ledger and ultimately append a new block to the chain in a moment.

Execute:
But first, go back to the query.js program and edit the constructor request to query CAR4. We do this by changing the function
in query.js from queryAllCars to queryCar and passing CAR7 as the specific key. Make sure you save!

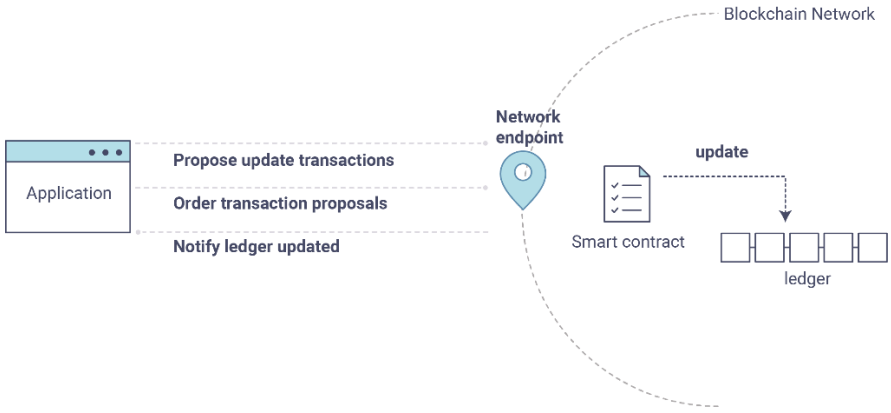
The query.js program should now look like this:



Execute:
Open the Integrated Terminal and go to fabcar directory and type
npm query.js



3. Updating the Ledger
Below we can see how this process works. An update is proposed, endorsed, then returned to the application, which in turn sends it to be ordered and written to every peer's ledger:



Our first update to the ledger will be to create a new car. We have a separate JavaScript program – invoke.js – that we will use to make updates. Just as with queries, use an editor to open the program and navigate to the code block where we construct our invocation:

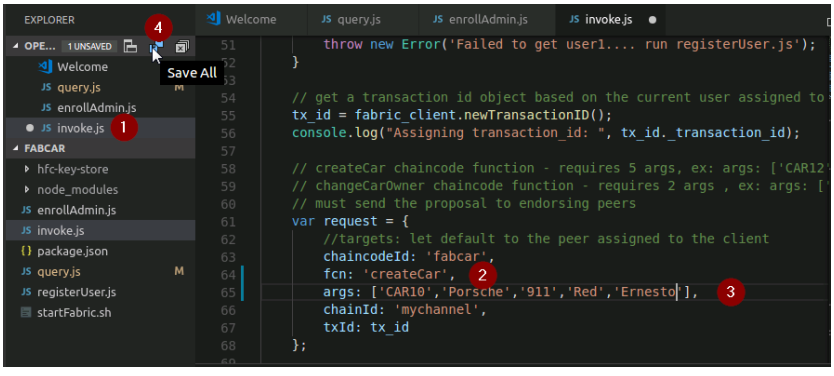
```
// createCar chaincode function - requires 5 args, ex: args: ['CAR12', 'Honda', 'Accord', 'Black', 'Tom'],

// changeCarOwner chaincode function - requires 2 args , ex: args: ['CAR10', 'Barry'],

// must send the proposal to endorsing peers

var request = {

  //targets: let default to the peer assigned to the client

  chaincodeId: 'fabcar',

  fcn: '',

  args: [''],

  chainId: 'mychannel',

  txId: tx_id

};
```

You'll see that we can call one of two functions - createCar or changeCarOwner. First, let's create a red Porsche 911 and give it to an owner named Ernesto. We're up to CAR9 on our ledger, so we'll use CAR10 as the identifying key here.

Execute:

Open invoke.js in VSCode and
Edit this code block to look like this:

```
var request = {
  //targets: let default to the peer assigned to the client
  chaincodeId: 'fabcar',
  fcn: 'createCar',
  args: ['CAR10', 'Porsche', '911', 'Red', 'Ernesto'],
  chainId: 'mychannel',
  txId: tx_id
};
```



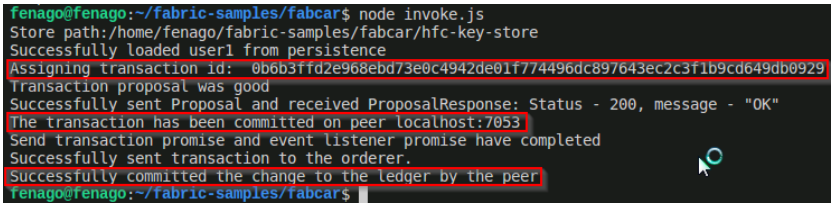
Save it and run the program:

node invoke.js

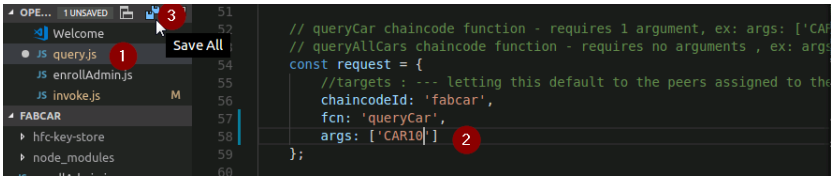
There will be some output in the terminal about Proposal

Response and promises. However, all we're concerned with is this message:

The transaction has been committed on peer localhost:7053



To see that this transaction has been written, go back to query.js and change the argument from CAR4 to CAR10.



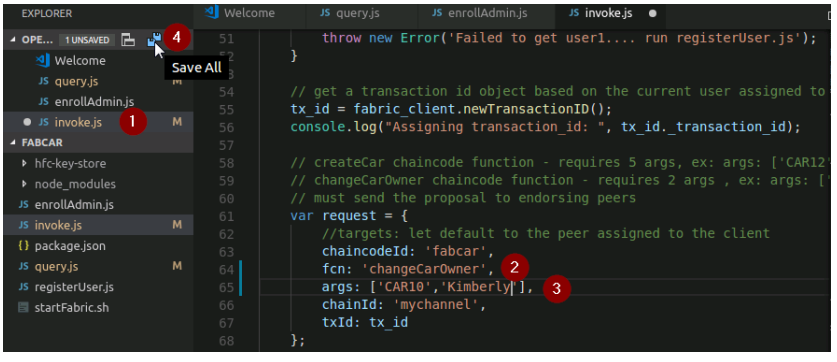
Execute
node query.js

```
fenago@fenago:~/fabric-samples/fabcar$ node query.js
Store path:/home/fenago/fabric-samples/fabcar/hfc-key-store
Successfully loaded user1 from persistence
Query has completed, checking results
Response is  {"colour":"Red","make":"Porsche","model":"911","owner":"Ernesto"}
fenago@fenago:~/fabric-samples/fabcar$
```

So now that we've done that, let's say that Ernesto is feeling generous and he wants to give his Porsche to someone named Kimberly.

To do this go back to invoke.js and change the function
from createCar to changeCarOwner and input the changes:

Execute:



The first argument – CAR10 – reflects the car that will be changing owners. The second argument – Kimberly – defines the new owner of the car.

Save and execute the program again:

node invoke.js

```
fenago@fenago:~/fabric-samples/fabcar$ node invoke.js
Store path:/home/fenago/fabric-samples/fabcar/hfc-key-store
Successfully loaded user1 from persistence
Assigning transaction_id:  157d931ae07c19bde2bd66dd02cce4487eeffbf96bc06d16e0ee0a0916a926b6
Transaction proposal was good
Successfully sent Proposal and received ProposalResponse: Status - 200, message - "OK"
The transaction has been committed on peer localhost:7053
Send transaction promise and event listener promise have completed
Successfully sent transaction to the orderer.
Successfully committed the change to the ledger by the peer
fenago@fenago:~/fabric-samples/fabcar$
```

Now let's query the ledger again and ensure that Kimberly is now associated with the CAR10 key:

node query.js

It should return this result:

```
fenago@fenago:~/fabric-samples/fabcar$ node query.js
Store path:/home/fenago/fabric-samples/fabcar/hfc-key-store
Successfully loaded user1 from persistence
Query has completed, checking results
Response is  {"colour":"Red","make":"Porsche","model":"911","owner":"Kimberly"}
fenago@fenago:~/fabric-samples/fabcar$
```

The ownership of CAR10 has been changed from Ernesto to Kimberly.

Lab: Implement the Chaincode Interface w/ Hyperledger Fabric 1.2

PREREQUISITE

- The following lab exercise requires Hyperledger Fabric revision 1.2.
- Clean up from the prior labs
- cd /home/fenago/fabric-tools
- ./stopFabric.sh
- ./teardownFabric.sh
- ./teardownAllDocker.sh
- docker ps (ensure no running containers)

OBJECTIVES

Demonstrate and implement the Chaincode interface. It is available in both Go and node.js. The OBJECTIVES of this section is to show how these methods work:

Init : called when a chaincode receives an instantiate or upgrade transaction so that the cc may perform initialization (to include applicationstate)

Invoke : processes transaction proposals

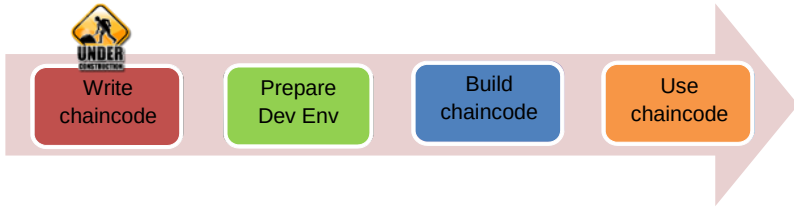
THEORY

Chaincode is a program, written in Go, node.js, that implements a prescribed interface. Eventually, other programming languages such as Java, will be supported. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it's similar to a "smart contract". A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a Query, which does not participate in state validation checks in subsequent commit phase.

In the following sections, we will explore chaincode through the eyes of an application developer. We'll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API.

LAB: Write Chaincode



Task 1: Write Chaincode

Execute:
In the Integrated Terminal (set to bash)

cd $GOPATH

cd src

mkdir -p btacc

cd btacc

touch btacc.go

```
fenago@fenago:~/go/src$ mkdir -p btacc
fenago@fenago:~/go/src$ cd btacc
fenago@fenago:~/go/src/btacc$ touch btacc.go
fenago@fenago:~/go/src/btacc$ ls
btacc.go
fenago@fenago:~/go/src/btacc$ pwd
/home/fenago/go/src/btacc
fenago@fenago:~/go/src/btacc$
```

Open btacc.go in VSCode and add the following code:

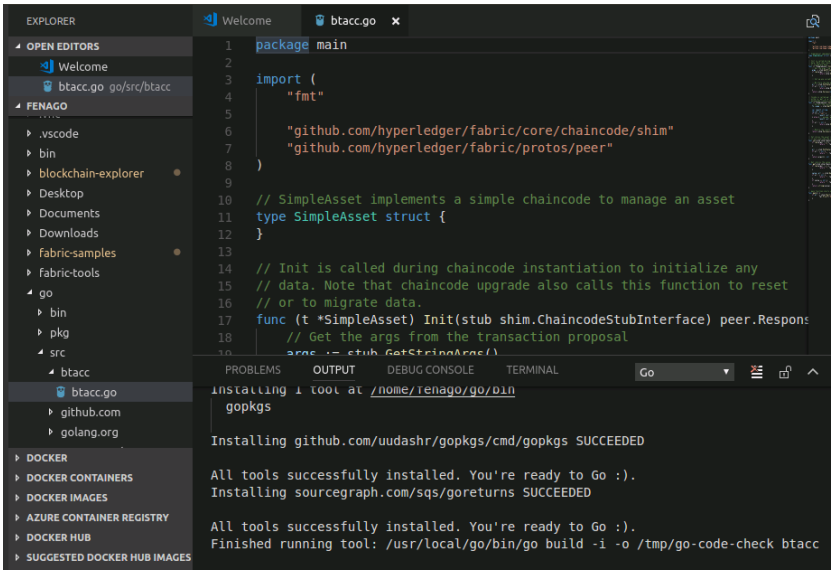
```
package main


import (
   "fmt"
```

```
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
}

// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
            return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
            return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}
```

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response
{
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else { // assume 'get' even if fn is nil
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}

// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
            return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }
```

```
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

// main function starts up the chaincode in the container during instantiate
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
        fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}
```

On save – you can install gopkgs and any other extensions suggested.



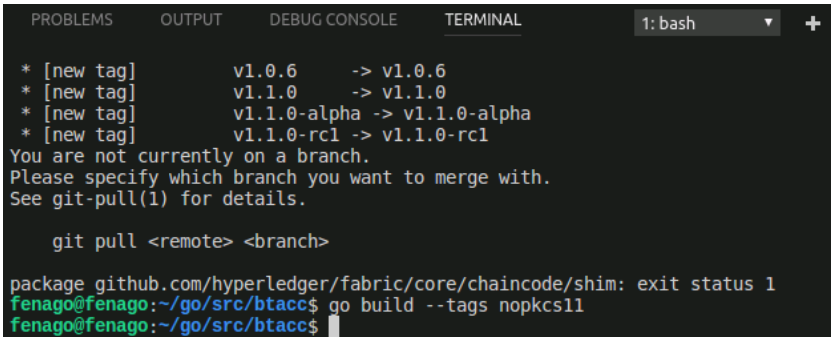
Close VSCode and reopen

If needed install golang-go (just type go at the command line)
cd ~
sudo apt install golang-go

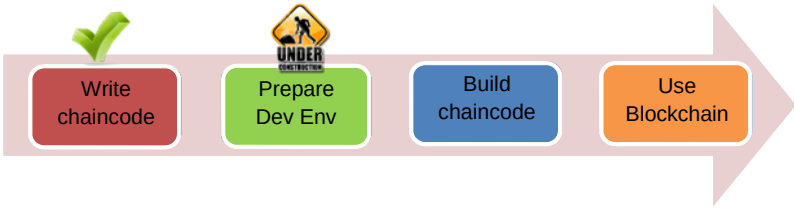
cd ~/go/src/btacc

go get -u --tags nopkcs11 github.com/hyperledger/fabric/core/chaincode/shim

go build –tags nopkcs11



Task 2 : Prepare the Development Environment



Make sure that you are inside of VSCode and using the Integrated terminal and execute:

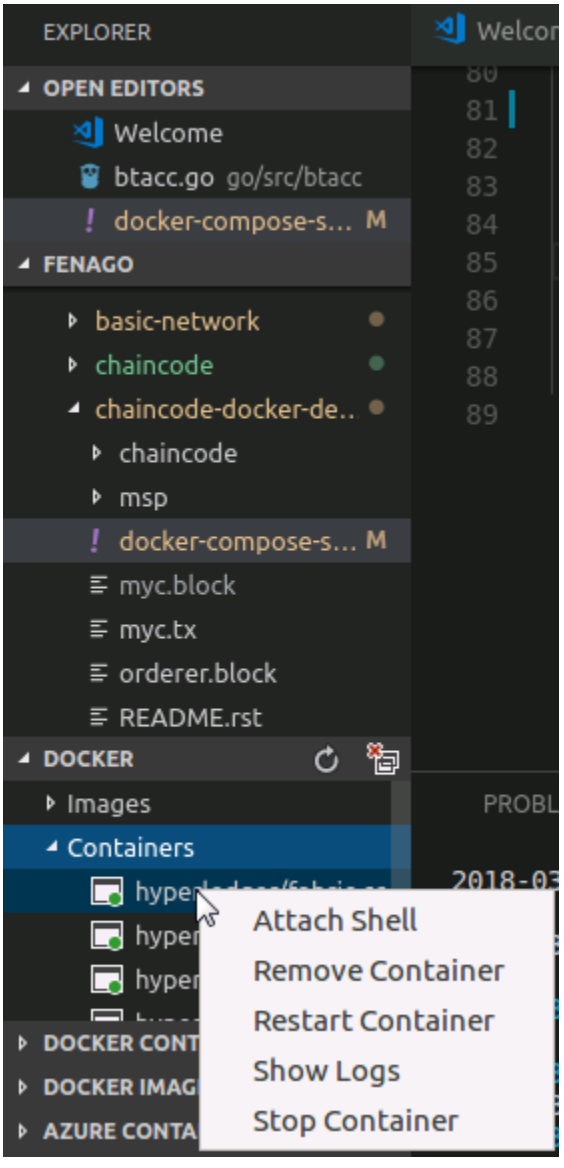
bash

cd /home/fenago/fabric-samples/chaincode-docker-devmode

docker-compose -f docker-compose-simple.yaml up

*if you notice any problems then stop all containers and restart

Right Click on the chaincode container in Docker Containers and select Attach Shell



Type bash to go into the bash shell

cd into the Simple Asset Chaincode:

cd sacc

build

Now register the chaincode:

CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc

```
root@43212d207325:/opt/gopath/src/chaincode# cd sacc
root@43212d207325:/opt/gopath/src/chaincode/sacc# ls
sacc.go
root@43212d207325:/opt/gopath/src/chaincode/sacc# go build
root@43212d207325:/opt/gopath/src/chaincode/sacc# CORE PEER ADDRESS=peer:7052 CORE CHAINCOD
E ID NAME=mycc:0 ./sacc
2018-03-26 01:14:45.324 UTC [shim] SetupChaincodeLogging -> INFO 001 Chaincode log level no
t provided; defaulting to: INFO
2018-03-26 01:14:45.326 UTC [shim] SetupChaincodeLogging -> INFO 002 Chaincode (build level
: ) starting up ...
```

Right Click on the CLI container and choose attach shell (you should now have 3 terminals running)

Inside of the container type bash to enter the bash shell.

Now you need to install and instantiate the chaincode

peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0

```
$ docker exec -it 2200eec25527093c10c5c48056180d12db3128abad60f2027a4b06de504ef048 /bin/sh
# bash
root@2200eec25527:/opt/gopath/src/chaincodedev# peer chaincode install -p chaincodedev/chai
ncode/sacc -n mycc -v 0
2018-03-26 01:17:28.991 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-03-26 01:17:28.993 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default s
igning identity
2018-03-26 01:17:28.993 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using defaul
t escc
2018-03-26 01:17:28.993 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using defaul
t vscc
```

```
2018-03-26 01:17:30.958 UTC [chaincodeCmd] install -> DEBU 00d Installed remotely response:
<status:200 payload:"OK" >
2018-03-26 01:17:30.959 UTC [main] main -> INFO 00e Exiting.....
```

peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc

```
root@2200eec25527:/opt/gopath/src/chaincodedev# peer chaincode instantiate -n mycc -v 0 -c
'{"Args":["a","10"]}' -C myc
2018-03-26 01:18:24.061 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-03-26 01:18:24.061 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default s
igning identity
2018-03-26 01:18:24.064 UTC [msp/identity] Sign -> DEBU 003 Sign: plaintext: 0AC3070A5B0801
1A0B08E094E1D50510...436F6E666967426C6F636B0A036D7963
2018-03-26 01:18:24.064 UTC [msp/identity] Sign -> DEBU 004 Sign: digest: AC9BF5E6918708CD5
091CADD486C6ABDC4A873D5CA510F2A59F2ADA43BA3450F
```

peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc

```
2018-03-26 01:31:42.764 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 065 Chaincode inv
oke successful. result: status:200 payload:"21"
2018-03-26 01:31:42.765 UTC [main] main -> INFO 066 Exiting.....
```

peer chaincode query -n mycc -c '{"Args":["query","e"]}' -C myc

```
Query Result: 21
2018-03-26 01:33:54.746 UTC [main] main -> INFO 008 Exiting
```

You will notice that we created btacc but we executed sacc. By default, we mount only sacc. However, you can easily test different chaincodes (btacc) by adding them to the chaincode subdirectory on the container and relaunching your network. At this point they will be accessible in your chaincode container.

Launch a new terminal and go into the bash shell

cd /home/fenago/go/src/btacc

peer chaincode install -n asset_mgmt -v 1.0 -p btacc

SUMMARY

This lab showed you how to interact with chaincode on HyperLedger.