HANDS-ON BLOCKCHAIN DEVELOPMENT

Hyperledger Fabric

Dr Cleverence Kombe (PhD)
Email: cleverence@gmail.com, twitter: @cleverence, linkeid linkedin.com/in/cleverence

Building Supply Chain Application with Hyperledger Fabric

In this session, we will design and build a simple supply chain blockchain application called **Pharma Ledger Network (PLN)**. This project will give you a taste of how blockchain enables global business transactions with greater transparency, streamlined supplier onboarding, better response to disruptions, and a secure environment. Specifically, the PLN project illustrates how blockchain can help manufacturers, wholesalers, and other supply chain members like pharmacies deliver medical supplies.

This training will help you achieve the following practical goals:

- Designing a blockchain supply chain
- Writing chaincode as a smart contract
- Compiling and deploying Fabric chaincode
- Running and testing the smart contract
- Developing an application with Hyperledger Fabric through the SDK

Designing a Blockchain Supply Chain

The traditional supply chain usually lacks transparency and reliable reporting. Large organizations have built their own systems to enable global control of their daily operations while recording transactions between suppliers and distributors in real time. However, many small companies lack that information and have limited visibility to trace their products at any given moment. That means, in their entire supply chain product process flow (from production to consumption), the transparency from upstream to downstream is very limited. This could lead to inaccurate reports and a lack of interoperability.

By design, the blockchain is a shared-ledger, transparent, immutable, and secure decentralized system. It is considered a good solution for traditional supply chain industries at registering, controlling, and transferring assets. Indeed, the popularity of blockchain and its adoption, in part, stems from its use in supply chain systems around the world.

A smart contract, which defines a business function, can be deployed in blockchain and then accessed by multiple parties in the blockchain network. Each member in the blockchain will be assigned unique identifiers to sign and verify the blocks they add to the blockchain. During the life cycle of the supply chain, when authorized members in a consortium network invoke a smart contract function, the state data will be updated, after which current assets' status and the transaction data will become a permanent record in the ledger. Likewise, the processes related to assets can be easily and quickly moved from one step to another. The digital transactions in the ledger can be tracked, shared, and queried by all supply chain participants in real time. It provides organizations with new opportunities to correct problems within their supply chain system as it revolves around a single source of truth.

In this section, we discuss a simple supply chain system and build our PLN use case. It will provide a good foundation for analysing and implementing an application based on Hyperledger Fabric. We will analyse the business process workflow, identify the organizations in the network, and then design the consortium network. We'll also define a smart contract function that each organization will perform.

Understanding the Supply Chain Workflow

Let's take a look at organizations in the PLN business scenario, as shown in ***Figure 1***. For demonstration purposes, we simplified the pharma ledger process, as it can be much more complex in the real world

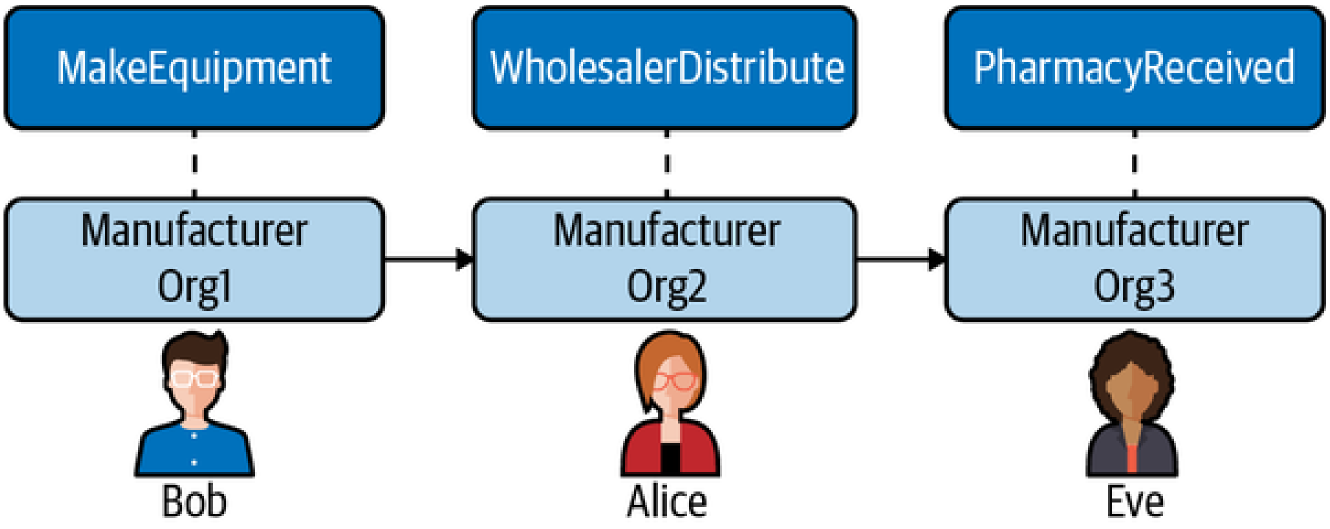


Figure 1: Organizations in the PLN

Our PLN process is divided into the following three steps:

1. A manufacturer makes equipment and ships it to the wholesaler.
2. A wholesaler distributes the equipment to the pharmacy.
3. The pharmacy, as a consumer, receives the equipment, and the supply chain workflow is completed.

Defining a Consortium

As we can see from the process workflow, our PLN involves three organizations: manufacturer, wholesaler, and pharmacy. These three entities will join together to build a consortium network to carry out the supply chain business. The consortium members can create users, invoke smart contracts, and query blockchain data. ***Table 1*** depicts the organizations and users in the PLN consortium.

Table 1: The PLN consortium

Organization name	User	MSP	Peer
Manufacturer	Bob	Org1MSP	*peer0.org1.example.com*
Wholesaler	Alice	Org2MSP	*peer0.org2.example.com*
Pharmacy	Eve	Org3MSP	*peer0.org3.example.com*

In our PLN consortium, each of the three organizations has a user, an MSP, and a peer. For the manufacturer organization, we have a user called Bob as an application user. Org1MSP is an MSP ID to load the MSP definition. We define AnchorPeers with the hostname peer0.org1.example.com to gossip communication across. Similarly, the wholesaler is the second organization, Alice is its application user, and its MSP ID is Org2MSP. Finally, Eve is the pharmacy organization user with Org3MSP.

With the organizations identified, we can define our Hyperledger Fabric network topology, as shown in ***Figure 2***.

Since installing and deploying PLN in multiple physical nodes may not be within the scope of this session, we define one peer with four organizations, representing the manufacturer, wholesaler, pharmacy, and orderer nodes.

The channel ***plnchannel*** provides a private communications mechanism used by the orderer and the other three organizations to execute and validate the transactions.

Reviewing the PLN Life Cycle

As we mentioned in the previous section, the PLN life cycle has three steps: the manufacturer makes equipment and ships to the wholesaler; the wholesaler distributes the equipment to the pharmacy; and finally, the pharmacy receives the equipment. The entire process can be traced by equipment ID. Let’s look at this in more detail.

A piece of equipment with equipment ID 2000.001 was made by a manufacturer on January 1, with equipment and other attributes and values, as shown in ***Figure 3***.

Equipment attributes and values

Here we define a unique identification equipment number to represent equipment. Each equipment item is owned by an equipment owner at a certain period of time. In our case, we define three owner types: manufacturer, wholesaler, and pharmacy.

When a manufacturer makes a piece of equipment and records it in the PLN, the transaction result shows the equipment with a unique identification number of 2000.001 in the ledger. The current owner is ***GlobalEquipmentCorp***. The current owner type and previous one are the same - manufacturer. The equipment was made on January 1, 2021. The `lastUpdated` entry is the date when the transaction was recorded in the PLN.

After a few weeks, the manufacturer ships the equipment to the wholesaler, and the equipment state will change, including ownership, previous and current owner type, and last update. Let’s take a look at which equipment states change, as shown in ***Figure 4***.

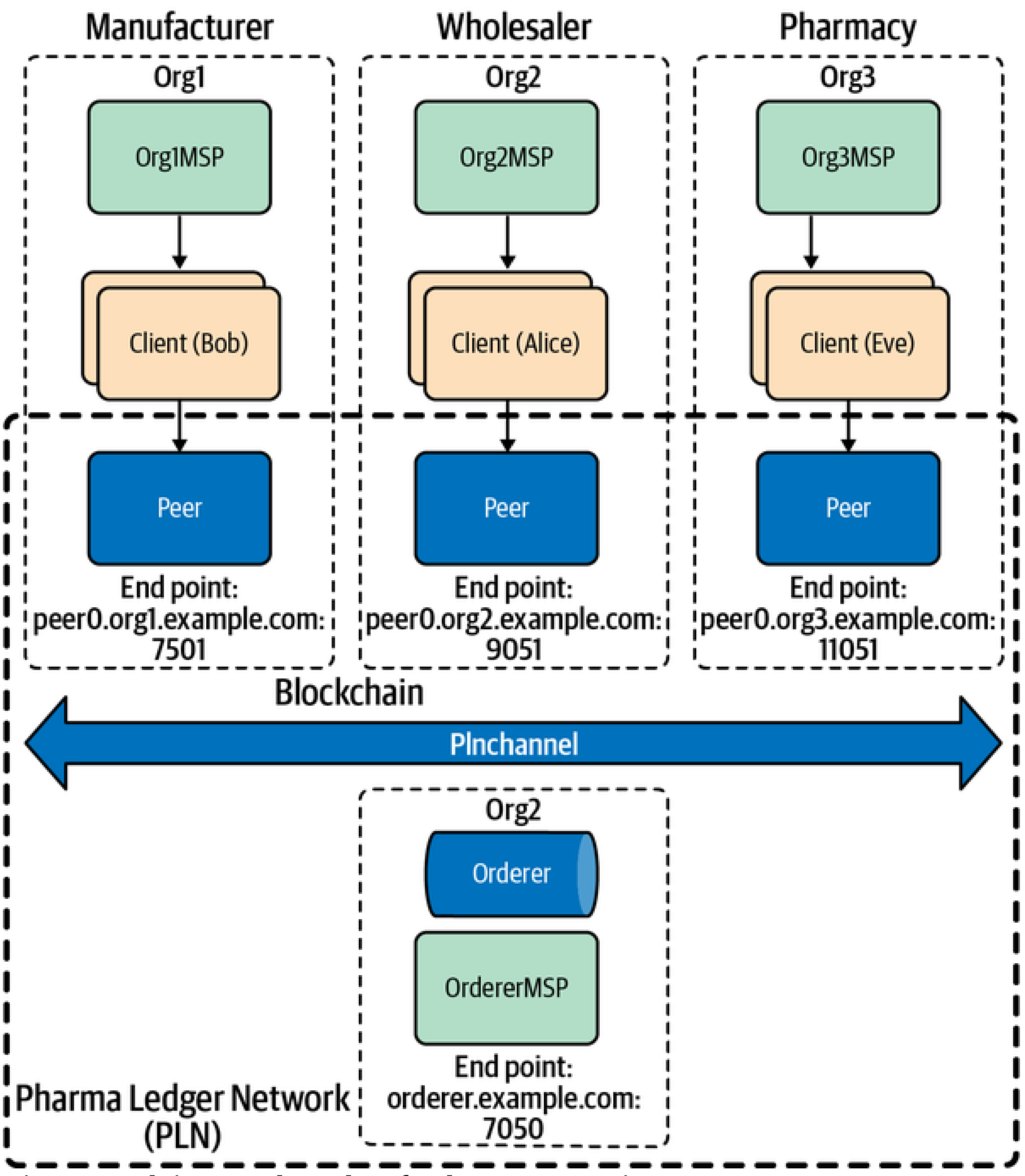
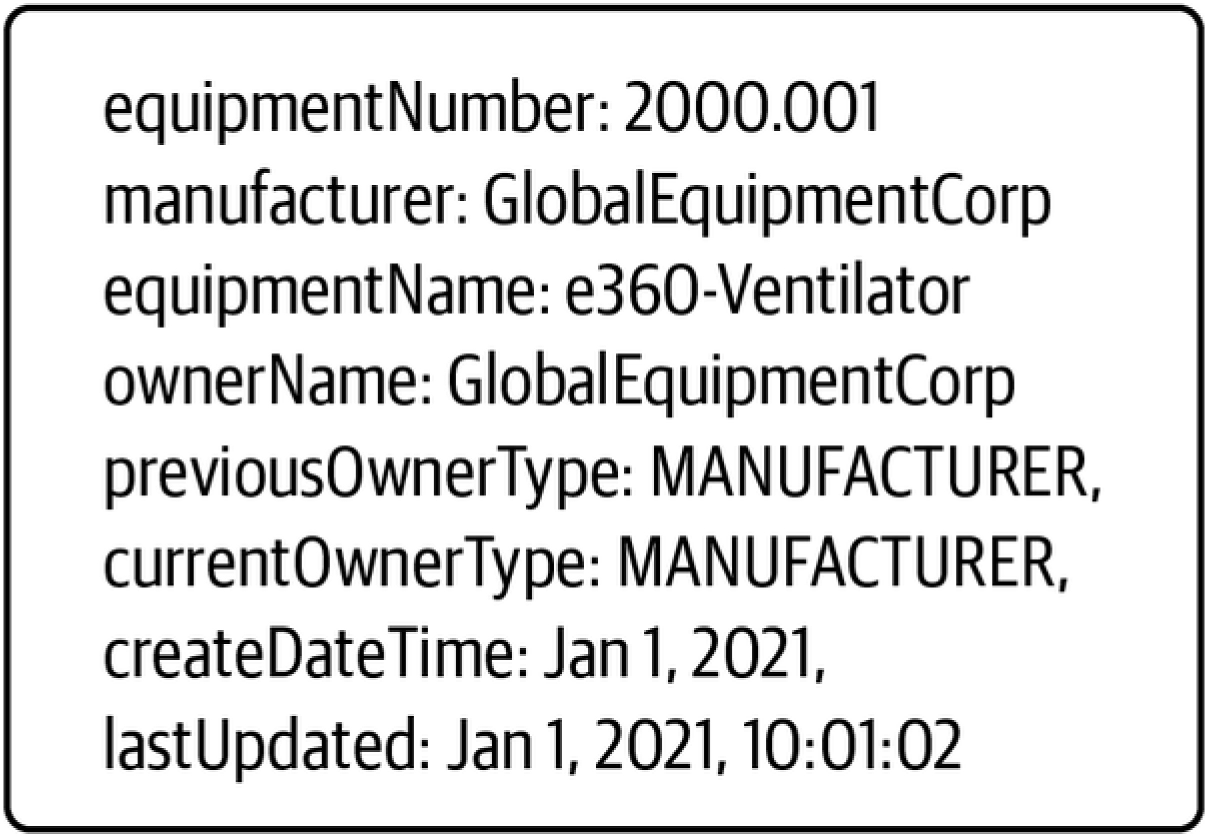
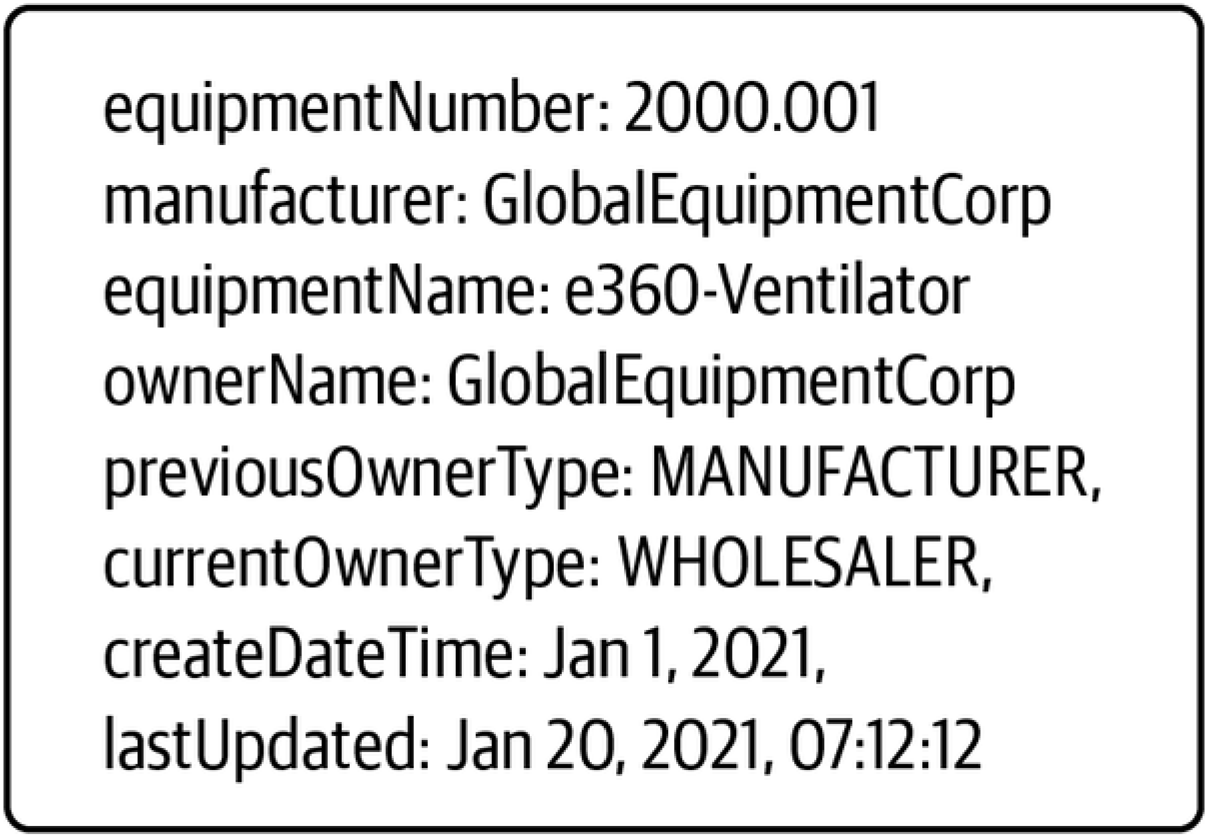

Manufacturer
Wholesaler
Pharmacy
Org1
Org2
Org3
Org1MSP
Org2MSP
Org3MSP
Client (Bob)
Client (Alice)
Client (Eve)
Peer
Peer
Peer
End point: peer0.org1.example.com: 7501
End point: peer0.org2.example.com: 9051
End point: peer0.org3.example.com: 11051
Blockchain
Plnchannel
Org2
Orderer
OrdererMSP
End point: orderer.example.com: 7050
Pharma Ledger Network (PLN)

Figure 2: Fabric network topology for the PLN consortium



```
equipmentNumber: 2000.001
manufacturer: GlobalEquipmentCorp
equipmentName: e360-Ventilator
ownerName: GlobalEquipmentCorp
previousOwnerType: MANUFACTURER,
currentOwnerType: MANUFACTURER,
createDateTime: Jan 1, 2021,
lastUpdated: Jan 1, 2021, 10:01:02
```

Figure 3: Equipment attributes and their values for the manufacturer



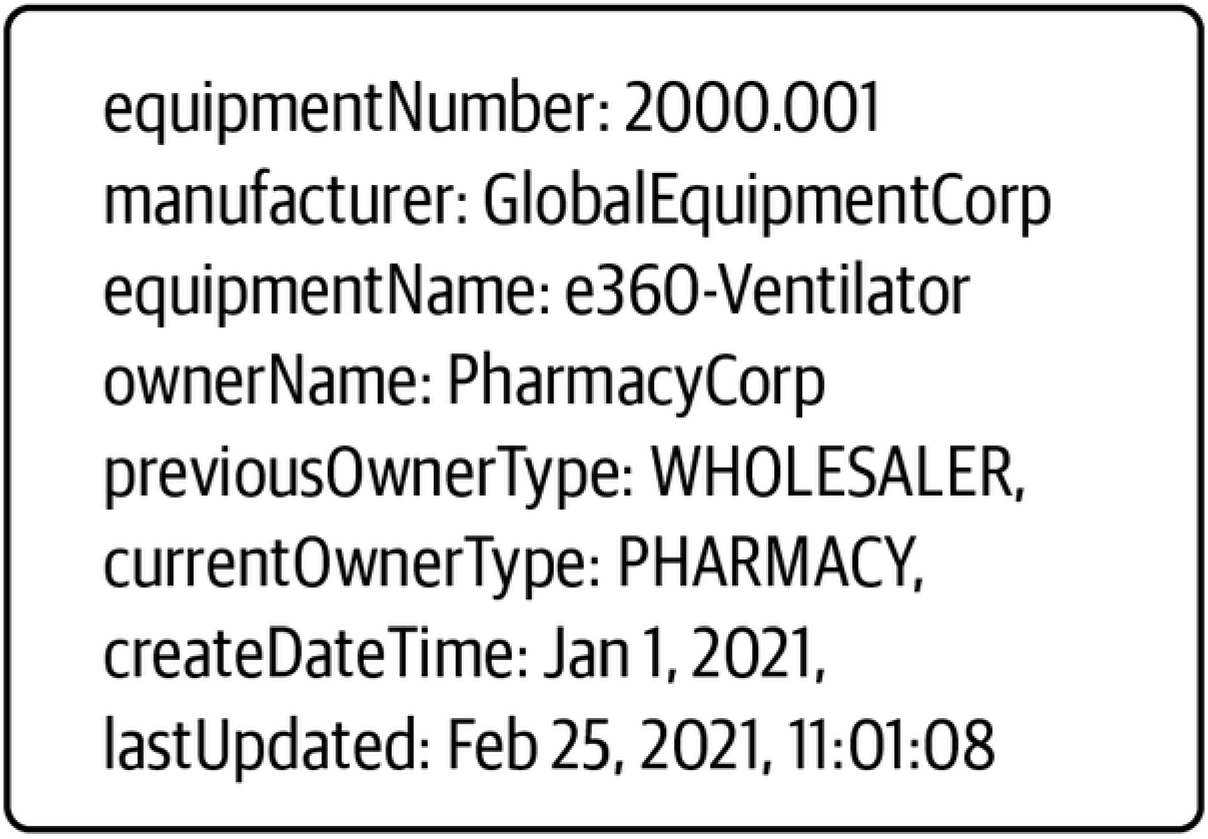
```
equipmentNumber: 2000.001
manufacturer: GlobalEquipmentCorp
equipmentName: e360-Ventilator
ownerName: GlobalEquipmentCorp
previousOwnerType: MANUFACTURER,
currentOwnerType: WHOLESALER,
createDateTime: Jan 1, 2021,
lastUpdated: Jan 20, 2021, 07:12:12
```

Figure 4: Equipment state changes for the wholesaler

Equipment state changes

One of the most significant changes is that the equipment is now owned by ***GlobalWholesalerCorp***. The previous owner type is the manufacturer. The last updated date has also changed.

After one month, the pharmacy finally receives this equipment order. The ownership is now transferred from the wholesaler to the pharmacy, as shown in ***Figure 5***. The supply chain flow can be considered closed.



```
equipmentNumber: 2000.001
manufacturer: GlobalEquipmentCorp
equipmentName: e360-Ventilator
ownerName: PharmacyCorp
previousOwnerType: WHOLESALER,
currentOwnerType: PHARMACY,
createDateTime: Jan 1, 2021,
lastUpdated: Feb 25, 2021, 11:01:08
```

Figure 5: Updated equipment values for the pharmacy

Equipment in the hand of the pharmacy

With the same equipment identity, the peer organization can trace the equipment's entire history of transaction records by looking up the equipment number.

Understanding Transactions

As you've seen, the entire life cycle has three steps. Originating from the manufacturer, the equipment moves from wholesaler to pharmacy. As such, as a result of making a piece of equipment, the wholesaler distributes and the pharmacy receives the transaction. With all of this design and analysis, we can now start to write our PLN smart contract.

Writing Chaincode as a Smart Contract

We have discussed how equipment state and attributes change during the life cycle of a transaction, as the equipment moves among parties in our Pharma Ledger Network. In Hyperledger Fabric, a smart contract is a program that implements the business logic and manages the world state of a business object during its life cycle. During deployment, this contract will be packaged into the chaincode and installed on each endorsing peer that runs in a secured Docker container. The Hyperledger Fabric smart contract can be programmed in Go, JavaScript, Java, and Python.

In this section, we will write a smart contract implementation for our PLN by using JavaScript. Also, we'll use Fabric v2.4.9 and Fabric CA v1.5.5 throughout the entire project.

Project Structure

To start our PLN smart contract development, first we need to create our smart contract project. Since we have three organizations, all peers must agree and approve of the new version of the smart contract that will be installed and deployed to the network. For our PLN, we will assume they are all the same:

We define a smart contract called *pharmaledgercontract.js*. The project structure is shown here:

```
+---manufacturer
|   +---application
|   |   |   app.js
|   |   |   package.json
|   |   +---public
|   |   +---services
|   |   \---views
|   |           index.ejs
|   \---contract
|       |   index.js
|       |   package.json
|       \---lib
|               pharmaledgercontract.js
```

The *package.json* file defines the two most important fabric libraries:

```
"dependencies": {
     "fabric-contract-api": "^2.4.9",
     "fabric-shim": "^2.4.9"
 },
```

fabric-contract-api provides the contract interface. It has two critical classes that every smart contract needs to use, `Contract` and `Context`:

```
const {Contract, Context} = require('fabric-contract-api');
```

`Contract` has `beforeTransaction`, `afterTransaction`, `unknownTransaction`, and `createContext` methods that are optional and overridable in the subclass. You can specify the JavaScript explicit contract class name by using its superclass to initialize itself.

The `Context` class provides the transactional context for every transactional invocation. It can be overridden for additional application behavior to support smart contract execution.

Contract Class

Our pharma ledger contract implementation will extend from the default built-in contract class from the *fabric-contract-api* library. Let's first define PharmaLedgerContract with a constructor: org.pln.PharmaLedgerContract gives a very descriptive name with a unique namespace for our contract. The unique contract namespace is important to avoid conflict when a shared system has many contracts from different users and operations:

```
const { Contract, Context } = require('fabric-contract-api');
class PharmaLedgerContract extends Contract {

    constructor() {
        super('org.pln.PharmaLedgerContract');
    }
}
```

Transaction Logic

As we discussed, PharmaLedgerContract will need three business functions to move the equipment owner from the manufacturer to the wholesaler, and finally pharmacy:

```
async makeEquipment(ctx, manufacturer, equipmentNumber, equipmentName,
ownerName) {
// makeEquipment logic
}
async wholesalerDistribute(ctx, equipmentNumber, ownerName) {
// wholesalerDistribute logic
}
async pharmacyReceived(ctx, equipmentNumber, ownerName) {
// pharmacyReceived logic
}
```

The manufacturer will be initialized, and an equipment entry is created. As you will notice, these functions accept a context as the default first parameter with equipment-related arguments (manufacturer, equipmentNumber, equipmentName, ownerName) from client input. When makeEquipment is called, the function expects four equipment attributes from the client and assigns it to new equipment:

```
async makeEquipment(ctx, manufacturer, equipmentNumber, equipmentName,
ownerName){
        let dt = new Date().toString();
        const equipment = {
            equipmentNumber,
            manufacturer,
            equipmentName,
            ownerName,
            previousOwnerType: 'MANUFACTURER',
            currentOwnerType: 'MANUFACTURER',
            createDateTime: dt,
            lastUpdated: dt
        };
await                                     ctx.stub.putState(equipmentNumber,
Buffer.from(JSON.stringify(equipment)));
}
```

At the end of makeEquipment, ctx.stub.putState will store the equipment's initial state value with the equipment number key on the ledger. The equipment JSON data will be stringified using JSON.stringify, then converted to a buffer. The buffer conversion is required by the shim API to communicate with the peer.

The function uses the JavaScript new Date to get the current date time and assign it to the lastUpdated date time. When transaction data is submitted, each peer will validate and commit a transaction.

After the equipment record is created by the manufacturer, the wholesaler and pharmacy will just need to update ownership to track the current owner. Both functions are similar:

```
async wholesalerDistribute(ctx, equipmentNumber, ownerName) {
        const equipmentAsBytes = await ctx.stub.getState(equipmentNumber);
        if (!equipmentAsBytes || equipmentAsBytes.length === 0) {
            throw new Error(`${equipmentNumber} does not exist`);
        }
        let dt = new Date().toString();
        const strValue = Buffer.from(equipmentAsBytes).toString('utf8');
        let record;
        try {
            record = JSON.parse(strValue);
            if(record.currentOwnerType!=='MANUFACTURER') {
  throw new Error(` equipment - ${equipmentNumber} owner must be MANUFACTURER`);
            }
            record.previousOwnerType= record.currentOwnerType;
            record.currentOwnerType = 'WHOLESALER';
            record.ownerName = ownerName;
            record.lastUpdated = dt;
       } catch (err) {
           throw new Error(`equipment ${equipmentNumber} data can't be
processed`);
       }
   await ctx.stub.putState(equipmentNumber,
Buffer.from(JSON.stringify(record)));
   }
```

In the wholesalerDistribute function, we query current equipment ledger data by calling ctx.stub.getState(equipmentNumber). Once data returns, we need to make sure equipmentAsBytes is not empty and equipmentNumber is a valid number. Since ledger data is in JSON string byte format, that data needs to convert encoded data to a readable JSON format by using Buffer.from().toString('utf8'). We then verify that the current equipment owner type is the manufacturer by using the returned data.

Once all these conditions are met, ctx.stub.putState is called again. The equipment owner state would be updated to the wholesaler with the current timestamp. But as an immutable transaction log, all historical changes of the world state will permanently store in the ledger. We will define the queryHistoryByKey function to query all this data in the next step.

The pharmacyReceived function is similar to wholesalerDistribute, so it needs to validate that the current owner is the wholesaler and then transfer ownership to the pharmacy before updating the equipment record:

```
if(record.currentOwnerType!=='WHOLESALER') {
throw new Error(` equipment - ${equipmentNumber} owner must be WHOLESALER`);
}
record.previousOwnerType= record.currentOwnerType;
record.currentOwnerType = 'PHARMACY';
```

Query Functions

After we implement all three equipment business functions, the ledger still needs a query function to search current equipment data, and a query history function to get all of the historical records.

ChaincodeStub is implemented by the *fabric-shim* library and provides GetState and GetHistoryForKey functions. In our case, the query definition is straightforward: we just need to call ctx.stub.getState to get the corresponding result.

GetHistoryForKey returns all historical transaction key values across time. We can iterate through these records and convert them to a JSON byte array and send the data back as a response. The timestamp tells us when the equipment state was updated. Each record contains a related transaction ID and a timestamp:

```
async queryHistoryByKey(ctx, key) {
     let iterator = await ctx.stub.getHistoryForKey(key);
     let result = [];
     let res = await iterator.next();
     while (!res.done) {
       if (res.value) {
         const obj = JSON.parse(res.value.value.toString('utf8'));
         result.push(obj);
       }
       res = await iterator.next();
     }
     await iterator.close();
     console.info(result);
     return JSON.stringify(result);
 }
```

That is all for the smart contract function we will implement for our PLN. Next, we will compile and deploy the Fabric chaincode.

Compiling and Deploying Fabric Chaincode

We have now successfully written our PLN chaincode using JavaScript. Before deploying our contract, we need to set up the Fabric network.
To get started with Hyperledger Fabric, we first need to meet some prerequisites. Please install Git, cURL, Node.js, npm, Docker and Docker Compose, and the Fabric installation script first.

Install Prerequisites

Before advancing any further, we need to install the following third-party tools:

- Linux (Ubuntu) / Windows / Mac OS
- Python
- Git
- cURL
- Docker and Docker Compose: Docker version 17.06.2-ce or greater is required.
- Go version 1.14.*x*
- Node.js runtime and npm: Node.js version 8 is supported (from 8.9.4 and higher). Node.js version 10 is supported (from 10.15.3 and higher).

To set up a network, we generate crypto material for an organization by using Cryptogen, create a consortium, and then bring up PLN with Docker Compose. Let’s first set up our project.

Review the Project Structure

We have defined all setup scripts and configuration files for our PLN project. The project structure is organized as follows:

```
|   loadFabric.sh
|---pharma-ledger-network
    |   net-pln.sh
    +---channel-artifacts
    +---configtx
    |       configtx.yaml
    +---docker
    +---organizations
    |   +---cryptogen
    |   +---manufacturer
    |   +---pharmacy
    |   +---wholesaler
    +---scripts
```

Let’s take a look at important configurations.

Cryptogen

Four crypto configurations are in the *cryptogen* folder for the orderer and the other three peer organizations. OrdererOrgs defines ordering nodes and creates an organization definition. PeerOrgs defines peers, organization, and managing peer nodes.

As we know, running components in the network requires a CA. The Fabric Cryptogen tool will use those four crypto configuration files to generate the required X.509 certificates for all organizations.

For OrdererOrgs, we define the following crypto configuration:

```
OrdererOrgs:
  - Name: Orderer
    Domain: example.com
    EnableNodeOUs: true
    Specs:
      - Hostname: orderer
        SANS:
          - localhost
```

For PeerOrgs, we define the following crypto configuration for Org1 (manufacturer). The other two orgs are similar:

```
PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true
    Template:
      Count: 1
      SANS:
        - localhost
    Users:
      Count: 1
```

We set EnableNodeOUs to true, which enables the identity classification.

Configtx

The *configtx.yaml* file will generate OrdererSystemChannelGenesis and related artifacts by *configtx.yaml* configuration. In the *configtx.yaml* Organizations section, we define OrdererOrg and the other three peer organizations - Org1, Org2, and Org3, representing manufacturer, wholesaler, and pharmacy, respectively. Each organization will define its Name, ID, MSPDir, and AnchorPeers. MSPDir describes Cryptogen-generated output MSP directories. AnchorPeers specifies the peer node's host and port. It updates transactions based on peer policy for communication between network organizations and finds all active participants of the channel:

```
Organizations:
    - &OrdererOrg
        Name: OrdererOrg
        ID: OrdererMSP
        MSPDir: ../organizations/ordererOrganizations/example.com/msp
        Policies:
            ....
        OrdererEndpoints:
            - orderer.example.com:7050
    - &Org1
        Name: Org1MSP
        ID: Org1MSP
        MSPDir: ../organizations/peerOrganizations/org1.example.com/msp
        Policies:
...
        AnchorPeers:
            - Host: peer0.org1.example.com
              Port: 7051
- &Org2
        AnchorPeers:
            - Host: peer0.org2.example.com
              Port: 9051
- &Org3
        AnchorPeers:
            - Host: peer0.org3.example.com
              Port: 11051
```

The Organization Policies section defines who needs to approve the organization resource. In PLN, we use signature policies. For example, we define the Org2 Readers policy next, which allows the Org2 admin, peer, and client to access the resource in this node and allows peers to do only transaction endorsement. You can define your own policy per your application's needs.

```
Policies:
            Readers:
                Type: Signature
                Rule: "OR('Org2MSP.admin', 'Org2MSP.peer', 'Org2MSP.client')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org2MSP.peer')"
```

The Profiles section defines how to generate PharmaLedgerOrdererGenesis, including order configuration and organizations in the PLN consortiums:

```
Profiles:
   PharmaLedgerOrdererGenesis:
       <<: *ChannelDefaults
       Orderer:
           <<: *OrdererDefaults
           Organizations:
               - *OrdererOrg
           Capabilities:
               <<: *OrdererCapabilities
       Consortiums:
           PharmaLedgerConsortium:
               Organizations:
                   - *Org1
                   - *Org2
                   - *Org3
   PharmaLedgerChannel:
       Consortium: PharmaLedgerConsortium
       <<: *ChannelDefaults
       Application:
           <<: *ApplicationDefaults
           Organizations:
               - *Org1
               - *Org2
               - *Org3
           Capabilities:
               <<: *ApplicationCapabilities
```

Docker

The *docker* folder contains the Docker Compose configuration file, *docker-compose-pln-net.yaml*. The Docker Compose tool uses this configuration file to initialize the Fabric runtime environment. It defines volumes, networks, and services.

In our PLN project, we define our network name as pln. We first need to specify the Docker runtime environment variable for each organization service. For example, we define our blockchain network name as ${COMPOSE_PROJECT_NAME}_pln. When we assign the environment variable COMPOSE_PROJECT_NAME a net value, the network name will be net_pln. The container pulls the orderer images as well from *hyperledger/fabric-peer.* The volume configuration maps the directories where MSP, TLS, and other organization Fabric parts are used in the environment settings. Finally, working_dir sets the working directory for the peer:

```
services:
 orderer.example.com:
   container_name: orderer.example.com
   image: hyperledger/fabric-orderer:$IMAGE_TAG
   environment:..
   working_dir: /opt/gopath/src/github.com/hyperledger/fabric
   command: orderer
   volumes:..
   ports:
     - 7050:7050
   networks:
     - pln

 peer0.org1.example.com:
   container_name: peer0.org1.example.com
   image: hyperledger/fabric-peer:$IMAGE_TAG
```

```
    environment:
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_pln
..
      - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
      - CORE_PEER_LISTENADDRESS=0.0.0.0:7051
      - CORE_PEER_CHAINCODEADDRESS=peer0.org1.example.com:7052
      - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
      - CORE_PEER_GOSSIP_BOOTSTRAP=peer0.org1.example.com:7051
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
      - CORE_PEER_LOCALMSPID=Org1MSP
    volumes:
...
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: peer node start
    ports:
      - 7051:7051
    networks:
      - pln
```

Install Binaries and Docker Images

We have reviewed important configurations in order to run the PLN network. The *net-pln.sh* script will bring up the PLN network, but we first need to download and install Fabric binaries to your system. Under the root project folder is a file called *loadFabric.sh*; run the following command to load Fabric binaries and configs:

```
./loadFabric.sh
```

This will install the Hyperledger Fabric platform-specific binaries and config files into the */bin* and */config* directories under the project. For this project, we use the current latest production releases: Fabric v2.4.9 and Fabric CA v1.5.5. Run docker images -a to check installed Fabric images.

Note

Make sure all of the scripts in the project are executable. For example, you can run **chmod +x loadFabric.sh** to make it executable.

It is now time to start our PLN network.

Start the PLN Network

As we mentioned before, to start the PLN network, we need to complete the following steps:

1. Generate peer organization certificates using the Cryptogen tool. Here is the command for Org1:

   ```
   Cryptogen  generate  --config=./organizations/cryptogen/crypto-config-
   org1.yaml --output="organizations"
   ```

   The generated output is stored in the *organizations* folder.

2. Create orderer organization identities using Cryptogen:

```
cryptogen generate --config=./organizations/cryptogen/crypto-config-orderer.yaml --output="organizations"
```

3. Generate a common connection profile (CCP) for Org1, Org2, and Org3:

```
./organizations/ccp-generate.sh
```

   ccp-generate.sh is under the *organizations* folder. It uses *ccp-template.json* and *ccp-template.yaml* files as templates; it defines placeholder variables for the org name, peer port, CA port, and CA PEM certificates. By passing these defined variables, we can generate org connection files when running *ccp-generate.sh*. And *ccp-generate.sh* will also copy generated connection files to the *peer orgs* folder:

```
echo "$(json_ccp $ORG $P0PORT $CAPORT $PEERPEM $CAPEM)" > organizations/peerOrganizations/org1.example.com/connection-org1.json

echo "$(yaml_ccp $ORG $P0PORT $CAPORT $PEERPEM $CAPEM)" > organizations/peerOrganizations/org1.example.com/connection-org1.yaml
```

   Each peer web client will use these connection files to connect to the Fabric network.

4. Create the consortium and generate an orderer system channel genesis block:

```
configtxgen -profile PharmaLedgerOrdererGenesis -channelID system-channel -outputBlock ./system-genesis-block/genesis.block
```

   configtxgen reads the *configtx.yaml* profile and generates the *genesis.block* file under the *system-genesis-block* folder.

5. Bring up the peer and orderer nodes.

   The *docker-compose* file is defined under *docker/docker-compose-pln-net.yaml*. The command will pull the latest Fabric orderer and peer images, build the orderer and peer images, and start the services we defined in the *.yaml* file. Run the following docker-compose command to bring up the peer and orderer nodes:

```
IMAGE_TAG=$IMAGETAG docker-compose ${COMPOSE_FILES} up -d 2>&1
```

6. Now, let's bring up the PLN network. Open a terminal window and run *net-pln.sh* under the *pharma-ledger-network* folder:

```
cd pharma-ledger-network

./net-pln.sh up
```

You should see the following success log:

```
Creating network "net_pln" with the default driver
Creating volume "net_orderer.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating volume "net_peer0.org3.example.com" with default driver
Creating orderer.example.com    ... done
Creating peer0.org2.example.com ... done
Creating peer0.org1.example.com ... done
Creating peer0.org3.example.com ... done
CONTAINER ID   IMAGE                              COMMAND   ..            NAMES
5a1fb5778a94       hyperledger/fabric-peer:latest     "peer   node   start"   ...
peer0.org3.example.com
969a5a9f5a85       hyperledger/fabric-peer:latest     "peer   node   start"   ...
peer0.org1.example.com
2f2cf2b0463d     hyperledger/fabric-peer:latest       "peer   node   start"   ...
peer0.org2.example.com
f327510667ff   hyperledger/fabric-orderer:latest   "orderer"                 ...
orderer.example.com
```

We have four organizations, including three peers and one orderer, that are running in the *net_pln* network. In the next step, we will use the script to create a PLN channel for all orgs.

Monitor the PLN Network

The Fabric images in the PLN network are Docker based. During project development or the production life cycle, you may encounter many errors. Log monitoring is one of the most important things to do from a DevOps standpoint for troubleshooting the code. It will help troubleshoot and find the root cause much easier and faster.

Logspout is an open source container log tool for monitoring Docker logs. It collects Docker's logs from all nodes in your cluster to be aggregated into one place. In our PLN project, we will use Logspout to monitor channel creation, smart contract installation, and other actions. Navigate to the *pharma-ledger-network* folder and open a new terminal window:

```
cd pharma-ledger-network
```

Run the following command from the *net-pln.sh* script and start the Logspout tool for the containers running on the PLN network *net_pln*:

```
./net-pln.sh monitor-up
```

```
...
Starting docker log monitoring on network 'net_pln'
Starting monitoring on all containers on the network net_pln
Unable to find image 'gliderlabs/logspout:latest' locally
latest: Pulling from gliderlabs/logspout
cbdbe7a5bc2a: Pull complete
956fa3cf18b6: Pull complete
94f24e0675e0: Pull complete
Digest: sha256:872555b51b73d7f50726baeae8d8c138b6b48b550fc71d733df7ffcadc9072e1
Status: Downloaded newer image for gliderlabs/logspout:latest
e8a8ad1787b69cfb7387264ee6ff63fd5a805aabe50ca6af6356d4cd8b27e052
```

Here is the script logic to bring up the Logspout tool; it pulls *gliderlabs/logspout* images by passing the PLN network name:

```
docker run -d --name="logspout" \

  --volume=/var/run/docker.sock:/var/run/docker.sock \

  --publish=127.0.0.1:${PORT}:80 \

  --network  ${DOCKER_NETWORK} gliderlabs/logspout
```

This terminal window will now show the PLN network container output for the remainder of the project development.

Tip

If you run into trouble during this process, check the Logspout terminal window to see errors.

Create a PLN Channel

To create the channel, we will use the configtxgen CLI tool to generate a genesis block, and then we'll use peer channel commands to join a channel with other peers. Creating a PLN channel requires several steps. All this script logic can be found in *scripts/createChannel.sh*:

1. Generate a channel configuration transaction file. In the *createChannel.sh* script, we define the createChannelTxn function. The critical command in this function is as follows:

   ```
   configtxgen -profile PharmaLedgerChannel -outputCreateChannelTx
   ./channel-artifacts/${CHANNEL_NAME}.tx -channelID $CHANNEL_NAME
   ```

   The configtxgen tool reads the profile PharmaLedgerChannel section from *configtx.yaml,* which defines channel-related configuration to generate the transaction and genesis block. It then generates the *plnchannel.tx* file.

2. Create an AnchorPeer configuration transaction file.

Next, we define the createAncorPeerTxn function. Similar to the previous step, we have defined the different organizational identities in *configtx.yaml*. The configtxgen tool reads the PharmaLedgerChannel organizational configuration and generates peer configuration transaction files:

```
configtxgen -profile PharmaLedgerChannel -outputAnchorPeersUpdate
./channel-artifacts/${orgmsp}anchors.tx -channelID $CHANNEL_NAME -asOrg
${orgmsp}
```

After createAncorPeerTxn runs, we should see the *Org1MSPanchors.tx*, *Org2MSPanchors.tx*, and *Org3MSPanchors.tx* transaction files generated.

3. Create a channel by using the peer channel command. The createChannel function uses the peer channel create command to create our PLN channel. When the command is issued, it will submit the channel creation transaction to the ordering service. The ordering service will check channel creation policy permissions defined in *configtx.yaml*. Only admin users can create a channel. The setGlobalVars function *scripts/utils.sh* will allow us to set the peer organization as the admin user. We use Org1 as an admin to create our channel.

   The commands are as follows:

```
setGlobalVars 1

peer channel create -o localhost:7050 -c $CHANNEL_NAME --
ordererTLSHostnameOverrideorderer.example.com -f ./channel-
artifacts/${CHANNEL_NAME}.tx --outputBlock ./channel-
```

setGlobalVars in *scripts/utils.sh* has the following logic for setting Org1 as an admin user. We can also use this function to set other peer organizations as admin users:

```
setGlobalVars() {
..
  if [ $USING_ORG -eq 1 ]; then
    export CORE_PEER_LOCALMSPID="Org1MSP"
    export CORE_PEER_TLS_ROOTCERT_FILE=$PEER0_ORG1_CA
    export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/
peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
..
}
```

4. After our PLN channel has been created, we can join all peers into this channel. joinMultiPeersToChannel in the *createChannel.sh* script will join all three peer orgs into our PLN channel by running the peer channel join command:

```
for org in $(seq 1 $TOTAL_ORGS); do
```

```
setGlobalVars $ORG

peer channel join -b ./channel-artifacts/$CHANNEL_NAME.block >&log.txt

done
```

When peer organizations join a channel, they need to be assigned as admin users by calling the `setGlobalVars` function and passing the `$ORG` parameter to it. The `peer channel join` command will use *genesis.block* to join peer orgs to the channel. Once the peer is joined to the channel, it can attend channel ledger block creation when receiving an ordering service transaction submission.

5. As the last step in the channel creation process, we need to select at least one peer as an anchor peer. An anchor peer's main role is private data and service discovery. The endpoints of the anchor peer are fixed. Other peer nodes belonging to different members can communicate with the anchor peers to discover all existing peers on a channel. To update an anchor peer, we set a selected peer as an admin user and issue a `peer channel update` command:

```
setGlobalVars $ORG

peer channel update -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com -c $CHANNEL_NAME -f ./channel-
artifacts/${CORE_PEER_LOCALMSPID}anchors.tx --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA

>&log.txt
```

You can use *net-pln.sh* to create the PLN channel by running the following command:

```
./net-pln.sh createChannel
```

Once channel creation is completed, you should see the following log:

```
...

***** [Step: 5]: start call updateAnchorPeers 3 on peer: peer0.org3,
channelID: plnchannel,
smartcontract: , version , sequence  *****
Using organization 3
2020-06-06 03:24:36.333 UTC [channelCmd] InitCmdFactory -> INFO 001
Endorser and orderer connections
initialized
2020-06-06 03:24:36.392 UTC [channelCmd] update -> INFO 002
Successfully submitted channel update
```

```
***** completed call updateAnchorPeers, updated peer0.org3 on
anchorPeers on channelID: plnchannel,
smartcontract: , version , sequence  *****

***** completed call updateOrgsOnAnchorPeers, anchorPeers updated on
channelID: plnchannel,
smartcontract: , version , sequence  *****

========= Pharma Ledger Network (PLN) Channel plnchannel successfully
joined ===========
```

To see the related container information, you can check the Logspout terminal window that we opened earlier.

Running and Testing the Smart Contract

We need to package a smart contract before we can install it to the channel. Navigate to the manufacturer contract folder directory and run the `npm install` command:

```
cd pharma-ledger-network/organizations/manufacturer/contract

npm install
```

This will install the *pharmaledgercontract* node dependency under *node_modules*.

Install the Smart Contract

Now we can start installing our smart contract by running the following *deploySmartContract.sh* script:

1. The `peer lifecycle chaincode package` command will package our smart contract. We assign the manufacturer as an administrator user to run the package command:

   ```
   setGlobalVars 1

   peer lifecycle chaincode package ${CHINCODE_NAME}.tar.gz --path
   ${CC_SRC_PATH} --lang

   ${CC_RUNTIME_LANGUAGE} --label ${CHINCODE_NAME}_${VERSION}
   ```

2. Install the chaincode on all peer orgs as an admin with the `peer lifecycle chaincode install` command:

   ```
   for org in $(seq 1 $CHAINCODE_ORGS); do

   setGlobalVars $ORG
   ```

```
peer lifecycle chaincode install ${CHINCODE_NAME}.tar.gz >&log.txt

done
```

When the chaincode package is installed, you will see messages similar to the following printed in your terminal:

```
2020-06-06 03:30:50.025 UTC [cli.lifecycle.chaincode]
submitInstallProposal -> INFO 001 Installed
remotely: response:<status:200
payload:"\nWpharmaLedgerContract_1:1940852a477d7697bb3a12d032268ff48c7
41c585db166403dd35f5e0b5c4e74\022
\026pharmaLedgerContract_1" >
2020-06-06 03:30:50.025 UTC [cli.lifecycle.chaincode]
submitInstallProposal -> INFO 002 Chaincode code
package identifier:
pharmaLedgerContract_1:1940852a477d7697bb3a12d032268ff48c741c585db1664
03dd35f5e0b5c4e74
***** completed call installChaincode, Chaincode is installed on
peer0.org1 on channelID: plnchannel,
smartcontract: pharmaLedgerContract, version 1, sequence 1 *****
```

3. After we install the smart contract, we need to query whether the chaincode is installed. We can query the packageID by using the peer lifecycle chaincode queryinstalled command:

```
peer lifecycle chaincode queryinstalled >&log.txt
```

If the command completes successfully, you will see logs similar to the following:

```
Installed chaincodes on peer:
Package ID: pharmaLedgerContract_1:
1940852a477d7697bb3a12d032268ff48c741c585db166403dd35f5e0b5c4e74,
Label: pharmaLedgerContract_1
***** completed call queryInstalled, Query installed successful with
PackageID is
pharmaLedgerContract_1:1940852a477d7697bb3a12d032268ff48c741c585db1664
03dd35f5e0b5c4e74
on channelID: plnchannel, smartcontract: pharmaLedgerContract, version
1, sequence 1 *****
```

4. With the returned package ID, we can now approve the chaincode definition for the manufacturer by using approveForMyOrg, which calls the peer lifecycle chaincode approveformyorg command:

```
peer lifecycle chaincode approveformyorg -o localhost:7050 --
ordererTLSHostnameOverride orderer.example.com --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA –channelID $CHANNEL_NAME -
-name ${CHINCODE_NAME} -- version ${VERSION} --package-id ${PACKAGE_ID}
--sequence ${VERSION} >&log.txt
```

5. We can check whether channel members have approved the same chaincode definition by using checkOrgsCommitReadiness, which runs the peer lifecycle chaincode checkcommitreadiness command:

   ```
   peer lifecycle chaincode checkcommitreadiness --channelID $CHANNEL_NAME
   --name ${CHINCODE_NAME} --version ${VERSION} --sequence ${VERSION} --
   output json >&log.txt
   ```

   As expected, we should see approvals for Org1MSP as true; the other two orgs are false:

   ```
   *****  [Step:  5]:  start  call  checkCommitReadiness  org1  on  peer:
   peer0.org1, channelID: plnchannel,
   smartcontract: pharmaLedgerContract, version 1, sequence 1 *****
   Attempting to check the commit readiness of the chaincode definition on
   peer0.org1, Retry after 3
   seconds.
   + peer lifecycle chaincode checkcommitreadiness --channelID plnchannel
   --name pharmaLedgerContract --
   version 1 --sequence 1 --output json
   {
           "approvals": {
                   "Org1MSP": true, "Org2MSP": false, "Org3MSP": false
           }
   }
   ```

6. The endorsement policy requires a set of majority organizations to endorse a transaction before it can commit the chaincode. We continue to run the peer lifecycle chaincode approveformyorg command for Org2 and Org3:

   ```
    ## approve org2
   approveForMyOrg 2
   ## check whether the chaincode definition is ready to be committed, two
   orgs should be approved
   checkOrgsCommitReadiness 3 1 1 0
   ## approve org3
   approveForMyOrg 3
   ## check whether the chaincode definition is ready to be committed, all
   3 orgs should be approved
   checkOrgsCommitReadiness 3 1 1 1
   ```

   If all commands execute successfully, all three orgs will approve the chaincode installation:

   ```
    {
           "approvals": {
                   "Org1MSP": true, "Org2MSP": true, "Org3MSP": true
           }
   }
   ***** completed call checkCommitReadiness, Checking the commit
   ```

```
readiness of the chaincode definition successful on peer0.org3 on
channel 'plnchannel' on channelID: plnchannel,
```

7. Now that we know for sure that the manufacturer, wholesaler, and pharmacy have all approved the *pharmaledgercontract* chaincode, we commit the definition. We have the required majority of organizations (three out of three) to commit the chaincode definition to the channel. Any of the three organizations can commit the chaincode to the channel by using the peer lifecycle chaincode commit command:

```
peer lifecycle chaincode commit -o localhost:7050 --
ordererTLSHostnameOverride orderer.example.com --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA --channelID $CHANNEL_NAME -
-name ${CHINCODE_NAME} $PEER_CONN_PARMS -- version ${VERSION} --
sequence ${VERSION} >&log.txt
```

8. We will use peer lifecycle chaincode querycommitted to check the chaincode commit status:

```
peer lifecycle chaincode querycommitted --channelID $ CHANNEL_NAME --
name ${CHINCODE_NAME} >&log.txt
```

9. Now that we've completed the chaincode deployment steps, let's deploy the *pharmaledgercontract* chaincode in our PLN network by running the following command:

```
./net-pln.sh deploySmartContract
```

If the command is successful, you should see the following response in the last few lines:

```
Committed chaincode definition for chaincode 'pharmaLedgerContract' on
channel 'plnchannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin:
vscc, Approvals:
[Org1MSP: true, Org2MSP: true, Org3MSP: true]
***** completed call queryCommitted, Query committed on channel
'plnchannel' on channelID: plnchannel,
smartcontract: pharmaLedgerContract, version 1, sequence 1 *****
***** completed call queryAllCommitted, Chaincode installed on
channelID: plnchannel, smartcontract: pharmaLedgerContract, version 1,
sequence 1 *****
=== Pharma Ledger Network (PLN) contract successfully deployed on
channel plnchannel  ====
```

After the chaincode is installed, we can start to invoke and test the chaincode methods for *pharmaledgercontract*.

Test the Smart Contract

We have created *invokeContract.sh* for this project. It defines an invocation method for makeEquipment, wholesalerDistribute, pharmacyReceived, and a query function. Now we can start testing our smart contract for these functions:

1. Call the makeEquipment chaincode method:

```
./net-pln.sh invoke equipment GlobalEquipmentCorp 2000.001 e360-Ventilator GlobalEquipmentCorp
```

2. We pass manufacturer, equipmentNumber, equipmentName, and ownerName as arguments. The script basically calls peer chaincode invoke commands by passing related function arguments:

```
peer chaincode invoke -o localhost:7050  --ordererTLSHostnameOverride
orderer.example.com –tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -
C    $CHANNEL_NAME    -n    ${CHINCODE_NAME}    $PEER_CONN_PARMS       -c
'{"function":"makeEquipment","Args":["'              $manufacturer'",
"'$equipmentNumber'", "' $equipmentName'","'$ownerName'"]}' >&log.txt
```

You will see logs similar to the following:

```
invokeMakeEquipment-->              manufacturer:GlobalEquipmentCorp,
equipmentNumber:2000.001, equipmentName: e360-
Ventilator,ownerName:GlobalEquipmentCorp
+ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls true
--cafile /home/ubuntu/Hyperledger-Fabric-V2/chapter7-supplychain/
pharma-ledger-
network/organizations/ordererOrganizations/example.com/orderers/ordere
r.example.com/msp/tlscacerts/tlsc
a.example.com-cert.pem -C plnchannel -n pharmaLedgerContract
--peerAddresses localhost:7051
--tlsRootCertFiles        /home/ubuntu/Hyperledger-Fabric-V2/chapter7-
supplychain/pharma-ledger-
network/organizations/peerOrganizations/org1.example.com/peers/peer0.o
rg1.example.com/tls/ca.crt --
peerAddresses            localhost:9051            --tlsRootCertFiles
/home/ubuntu/Hyperledger-Fabric-V2/chapter7-
supplychain/pharma-ledger-
network/organizations/peerOrganizations/org2.example.com/peers/peer0.o
rg2.example.com/tls/ca.crt --
peerAddresses            localhost:11051           --tlsRootCertFiles
/home/ubuntu/Hyperledger-Fabric-V2/chapter7-
supplychain/pharma-ledger-
network/organizations/peerOrganizations/org3.example.com/peers/peer0.o
rg3.example.com/tls/ca.crt -c
```

```
'{"function":"makeEquipment","Args":["GlobalEquipmentCorp","2000.001",
"e360-Ventilator",
"GlobalEquipmentCorp"]}'
2020-06-06 03:43:09.186 UTC [chaincodeCmd] chaincodeInvokeOrQuery ->
INFO 001 Chaincode invoke
successful. result: status:200
```

3. After invoking makeEquipment, we can run a query function to verify the ledger result. The query function uses the peer chaincode query command:

```
peer chaincode query -C $CHANNEL_NAME - n ${CHINCODE_NAME} -c
'{"function":"queryByKey","Args":["'$QUERY_KEY'"]}' >&log.txt
```

4. Issue the following script command to query equipment:

```
./net-pln.sh invoke query 2000.001
```

The query should return current equipment state data:

```
{"Key":"2000.001","Record":{"equipmentNumber":"2000.001",
"manufacturer":"GlobalEquipmentCorp","equipmentName":
"e360-Ventilator","ownerName":"GlobalEquipmentCorp",
"previousOwnerType":"MANUFACTURER","currentOwnerType":"MANUFACTURER",
"createDateTime":"Sat Jun 06 2020 03:43:09 GMT+0000 (Coordinated
Universal Time)","lastUpdated":"Sat Jun 06 2020 03:43:09 GMT+0000
(Coordinated Universal Time)"}}
```

5. Continue to invoke the remaining equipment functions for the wholesaler and pharmacy:

```
./net-pln.sh invoke wholesaler 2000.001 GlobalWholesalerCorp


./net-pln.sh invoke pharmacy 2000.001 PharmacyCorp
```

6. Once equipment ownership is moved to the pharmacy, the supply chain reaches its final state. We can issue queryHistoryByKey from the peer chaincode query command. Let's check equipment historical data:

```
./net-pln.sh invoke queryHistory 2000.001
```

We can see the following output in the terminal:

```
 ***** start call chaincodeQueryHistory on peer: peer0.org1, channelID:
plnchannel, smartcontract:
pharmaLedgerContract, version 1, sequence 1 *****
+ peer chaincode query -C plnchannel -n pharmaLedgerContract -c
'{"function":"queryHistoryByKey","Args":["2000.001"]}'
```

```
[{"equipmentNumber":"2000.001","manufacturer":"GlobalEquipmentCorp",
"equipmentName":"e360-Ventilator","ownerName":"PharmacyCorp",
"previousOwnerType":"WHOLESALER","currentOwnerType":"PHARMACY",
"createDateTime":"Sat Jun 06 2020 03:43:09 GMT+0000
(Coordinated Universal Time)","lastUpdated":"Sat Jun 06 2020 03:48:48
GMT+0000 (Coordinated Universal Time)"},{"equipmentNumber":"2000.001",
"manufacturer":"GlobalEquipmentCorp","equipmentName":"e360-
Ventilator",
"ownerName":"GlobalWholesalerCorp","previousOwnerType":"MANUAFACTURER"
,
"currentOwnerType":"WHOLESALER","createDateTime":"Sat Jun 06 2020
03:43:09 GMT+0000 (Coordinated Universal Time)","lastUpdated":"Sat
Jun 06 2020 03:46:41 GMT+0000 (Coordinated Universal Time)"},
{"equipmentNumber":"2000.001","manufacturer":"GlobalEquipmentCorp",
"equipmentName":"e360-Ventilator","ownerName":"GlobalEquipmentCorp",
"previousOwnerType":"MANUAFACTURER","currentOwnerType":"MANUAFACTURER"
,
"createDateTime":"Sat Jun 06 2020 03:43:09 GMT+0000 (Coordinated
Universal  Time)","lastUpdated":"Sat  Jun  06  2020  03:43:09  GMT+0000
(Coordinated Universal Time)"}]
***** completed call chaincodeQuery, Query History successful on
channelID: plnchannel,
smartcontract: pharmaLedgerContract, version 1, sequence 1 *****
```

All of the transaction history records are displayed as output. We have tested our smart contract, and it works as expected.

Developing an Application with Hyperledger Fabric Through the SDK

We just deployed our Pharma Ledger Network in the Fabric network. The next step is to build a Pharma Ledger client application to interact with the smart contract function in the network. Let's take a moment to examine the application architecture.

At the beginning of our PLN network section, we generated a CCP for Org1, Org2, and Org3. We will use these connection files to connect to our PLN network for each peer org. When the manufacturer application's user Bob submits a makeEquipment transaction to the ledger, the pharma-ledger process flow starts. Let's quickly examine how our application works (***Figure 6***).

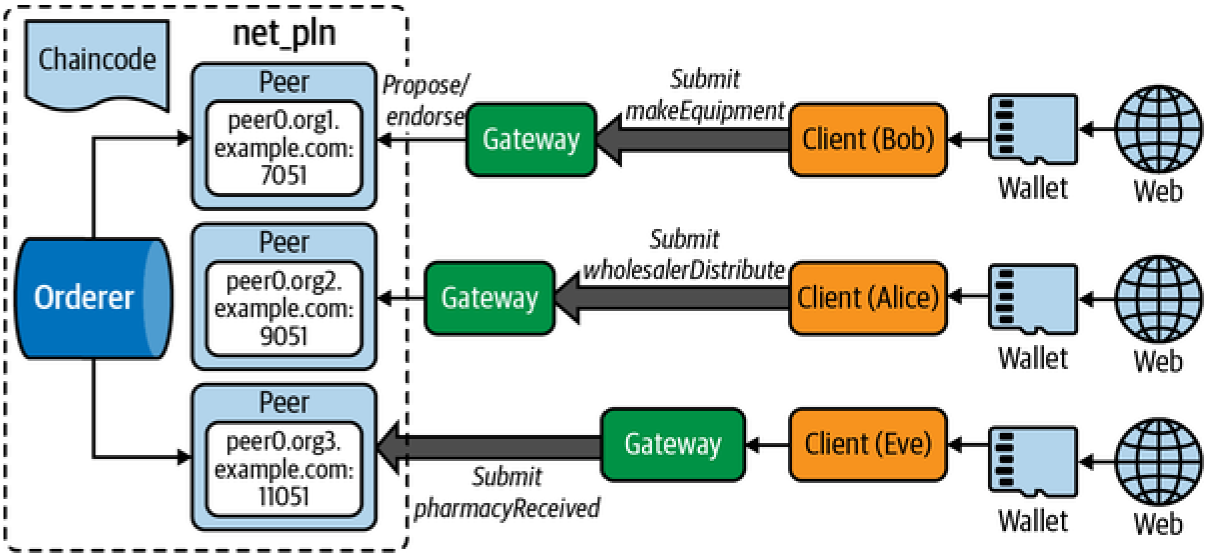


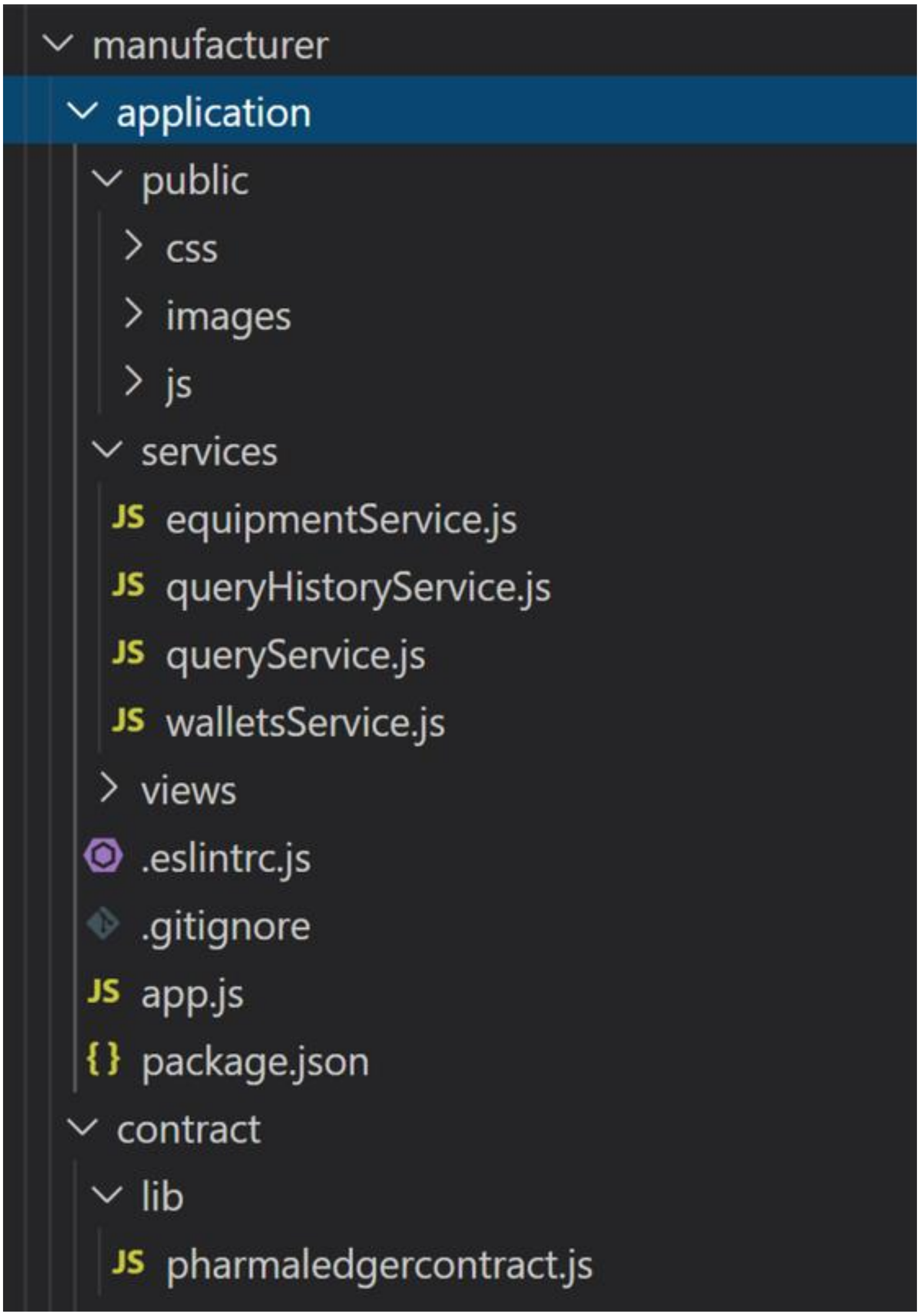
Figure 6: How the PLN application works

The manufacturer web user Bob connects to the Fabric network through a wallet. A wallet provides users an authorized identity that will be verified by the blockchain network to ensure access security. The Fabric SDK then submits a makeEquipment transaction proposal to *peer0.org1.example.com*. Endorsing peers verify the signature, simulate the proposal, and invoke the makeEquipment chaincode function with required arguments.

The transaction is initiated after the proposal response is sent back to the SDK. The application collects and verifies the endorsements until the endorsement policy of the chaincode is satisfied with producing the same result. The client then broadcasts the transaction proposal and proposal response to the ordering service.

The ordering service orders them chronologically by channel, creates blocks, and delivers the blocks of transactions to all peers on the channel. The peers validate transactions to ensure that the endorsement policy is satisfied and to ensure that no changes have occurred to the ledger state since the proposal response was generated by the transaction execution. After successful validation, the block is committed to the ledger, and world states are updated for each valid transaction.

You now understand the transaction end-to-end workflow. It is time to start building our pharma-ledger client application. ***Figure 7*** shows the application client project structure. The same folder structure is available for the wholesaler and pharmacy.



```
manufacturer
  application
    public
      css
      images
      js
    services
      equipmentService.js
      queryHistoryService.js
      queryService.js
      walletsService.js
    views
    .eslintrc.js
    .gitignore
    app.js
    package.json
  contract
    lib
      pharmaledgercontract.js
```

Figure 7: The application client project structure

We use *express.js* to build our node application. Let's review some important files.

The *package.json* file defines two Fabric-related dependencies:

```
    "dependencies" : {
       "fabric-contract-api" : "^2.4.9",
       "fabric-shim" : "^2.4.9"
   }
```

app.js defines all entry points for the manufacturer, and addUser will add a client user for the manufacturer, which in our case is Bob. makeEquipment will create equipment records when the manufacturer is an owner. queryByKey and queryHistoryByKey are common functions for all three orgs. The wholesaler and pharmacy will have similar functions:

```
app.post('/addUser',  async (req, res, next) => {
});
app.post('/makeEquipment',  async (req, res, next) => {
})
app. get('/queryHistoryByKey',  async (req, res, next) => {
})
app. get('/queryByKey', async (req, res, next) => {
})
```

addUser will call walletsService to add a user. Let's take a look at addToWallet(user) in walletsService:

```
const wallet = await
Wallets.newFileSystemWallet('../identity/user/'+user+'/wallet');
```

newFileSystemWallet will create a wallet for an input user (Bob) under the provided filesystem directory. Next, we find the user certificate and privateKey and generate an X.509 certificate to be stored in the wallet:

```
const credPath = path. join(fixtures,
'/peerOrganizations/org1.example.com/users/User1@org1.example.com');
const      certificate      =      fs.readFileSync(path.      join(credPath,
'/msp/signcerts/User1@org1.example.com-
cert.pem')).toString();
const       privateKey        =        fs.readFileSync(path.join(credPath,
'/msp/keystore/priv_sk')).toString();
```

The wallet calls key class methods to manage X509WalletMixin.createIdentity, which is used to create an Org1MSP identity using X.509 credentials. The function needs three inputs: mspid, the certificate, and the private key:

```
const identityLabel = user;
const identity = {
           credentials: {
               certificate,
               privateKey
           },
           mspId: 'Org1MSP',
            type: 'X.509'
}
const response = await wallet.put(identityLabel, identity);
```

Users from the manufacturer will call the equipmentService makeEquipment function. Before any user can call any of the smart contract functions, it needs to be authorized. To authorize user access to the blockchain, we need to follow these steps:

1. Find the user wallet created by adding a user function:

```
const wallet = await
Wallets.newFileSystemWallet('../identity/user/'+userName+'/wallet');
```

2. Load the connection profile associated with the user. Then the wallet will be used to locate and connect to a gateway:

```
const gateway =  new Gateway();

   let connectionProfile =

yaml.safeLoad(fs.readFileSync('../../../organizations/peerOrganization
s/org1.example.com/connection-org1.json', 'utf8'));

    // Set connection options; identity and wallet

let connectionOptions = {

  identity: userName,

wallet: wallet,

   discovery: { enabled: true, asLocalhost: true }

    };

      await gateway.connect(connectionProfile, connectionOptions);
```

3. Once a gateway is connected to a channel, we can find our pharmaLedgerContract with a unique namespace when creating a contract:

```
const network = await gateway.getNetwork('plnchannel');

const  contract  =  await  network.getContract('pharmaLedgerContract',
'org.pln.PharmaLedgerContract');
```

4. Submit the makeEquipment chain code invocation:

```
const response = await contract.submitTransaction('makeEquipment',

manufacturer, equipmentNumber, equipmentName, ownerName);
```

5. To verify that equipment records are stored in the blockchain, we can use Fabric query functions to retrieve the result. The following code shows how we can submit a query or queryHistory function to get equipment results:

```
const response = await contract.submitTransaction('queryByKey', key);
```

```
const response = await contract.submitTransaction('queryHistoryByKey',
key);
```

6. Let's bring up a manufacturer, create the user Bob, and then submit a transaction to our PLN blockchain. Navigate to the *pharma-ledger-network/organizations/manufacturer/application* folder and run `npm install`. When we start the application, we also make sure to update the client IP address in *plnClient.js* under *public/js*:

   ```
   var urlBase = "http://your-machine-public-ip:30000";
   ```

   ```
   npm install
   ```

   ```
   pharma-ledger-network/organizations/manufacturer/application$
   ```

   ```
   node app.js
   ```

   ```
   App listening at http://:::30000
   ```

7. In the manufacturer, we define the application port as 30000:

   Note

   Make sure this port is open, or you can change it to another available port number under the `app.js` line.

   ```
    var port = process.env.PORT || 30000;
   ```

8. Open a browser and enter `http://`*`your-machine-public-ip`*`:30000`:

   We will see the screen shown in ***Figure 8***.

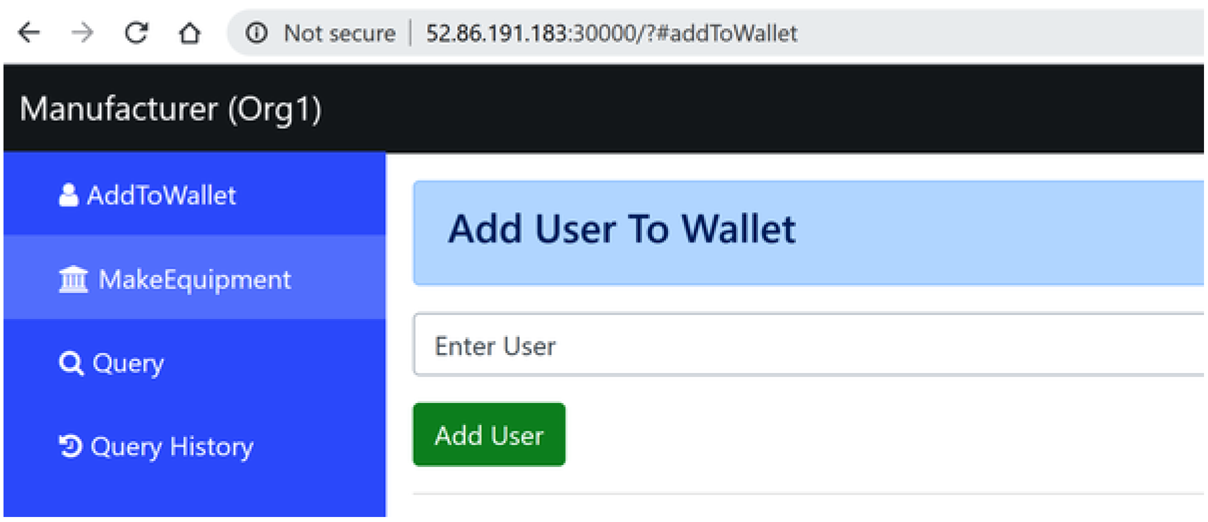
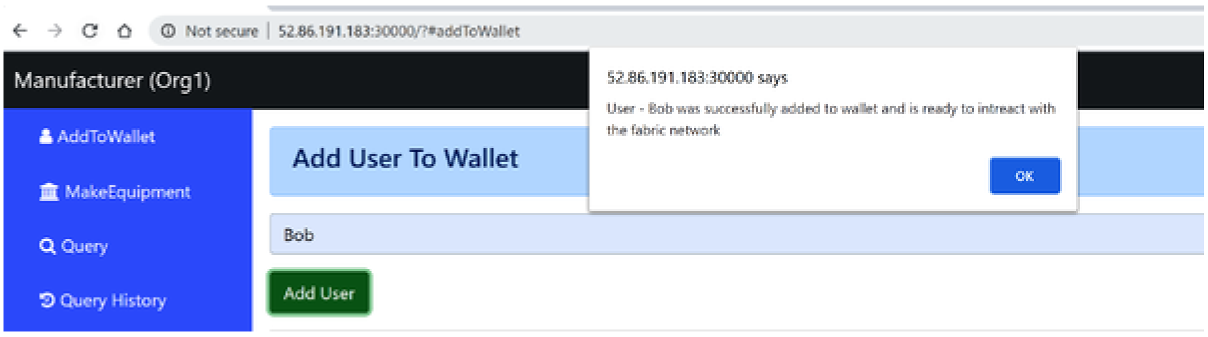


Figure 8: Adding a user to the wallet for the manufacturer

9. The default page is *addToWallet*. Since we have not added any user to the wallet so far, you can't submit makeEquipment and query history transactions at this moment. You have to add a user to the wallet. Let's add Bob as a manufacturer user, as shown in ***Figure 9***.

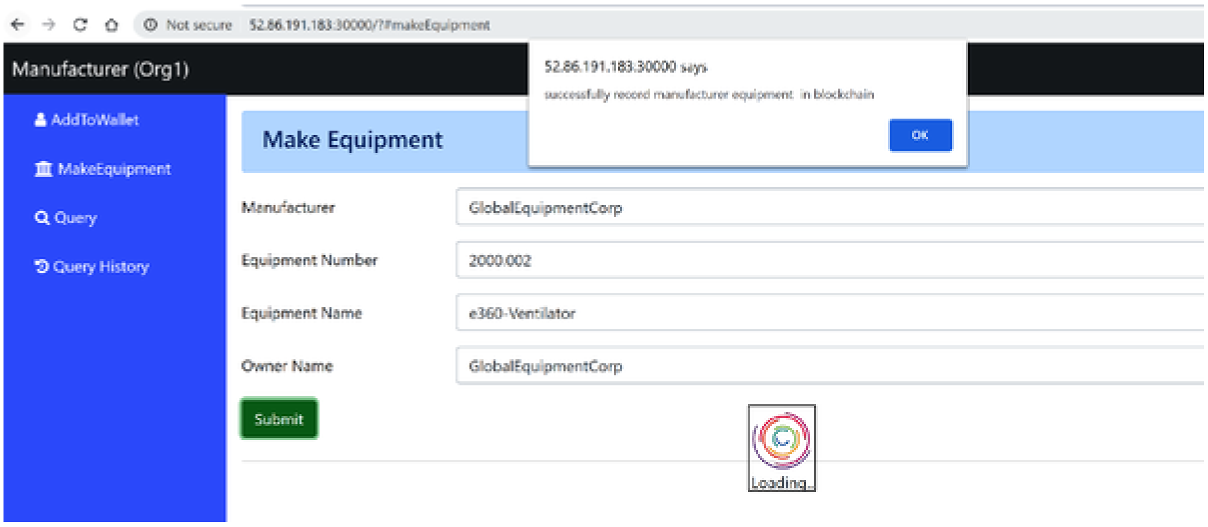


52.86.191.183:30000 says

User - Bob was successfully added to wallet and is ready to intreact with the fabric network

Figure 9: New user (Bob) is added

10. With the user wallet set up, the application can now connect to our PLN and interact with the chaincode. Click MakeEquipment on the left menu, enter all required equipment information, and submit the request (***Figure 10***). The success response will be returned from the blockchain.

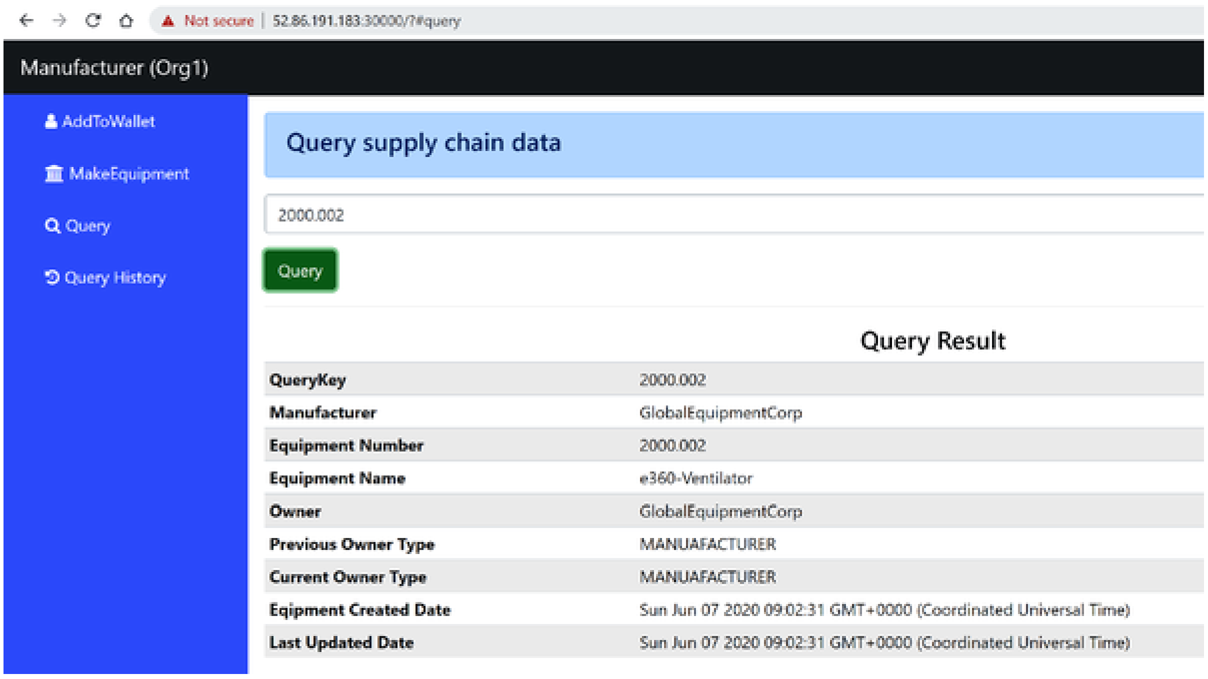


52.86.191.183:30000 says
successfully record manufacturer equipment in blockchain

Manufacturer	GlobalEquipmentCorp
Equipment Number	2000.002
Equipment Name	e360-Ventilator
Owner Name	GlobalEquipmentCorp

Figure 10: Adding equipment to the PLN network

11. We can now query equipment data in the PLN network by equipment number. ***Figure 11*** shows the result.



Query Result

QueryKey	2000.002
Manufacturer	GlobalEquipmentCorp
Equipment Number	2000.002
Equipment Name	e360-Ventilator
Owner	GlobalEquipmentCorp
Previous Owner Type	MANUAFACTURER
Current Owner Type	MANUAFACTURER
Eqipment Created Date	Sun Jun 07 2020 09:02:31 GMT+0000 (Coordinated Universal Time)
Last Updated Date	Sun Jun 07 2020 09:02:31 GMT+0000 (Coordinated Universal Time)

Figure 11: Query equipment on PLN network

12. Now open two other terminal windows, which will bring up node servers for the wholesaler and pharmacy, respectively.

    Navigate to */pharma-ledger-network/organizations/wholesaler/contract.* Run npm install to install the smart contract dependency first

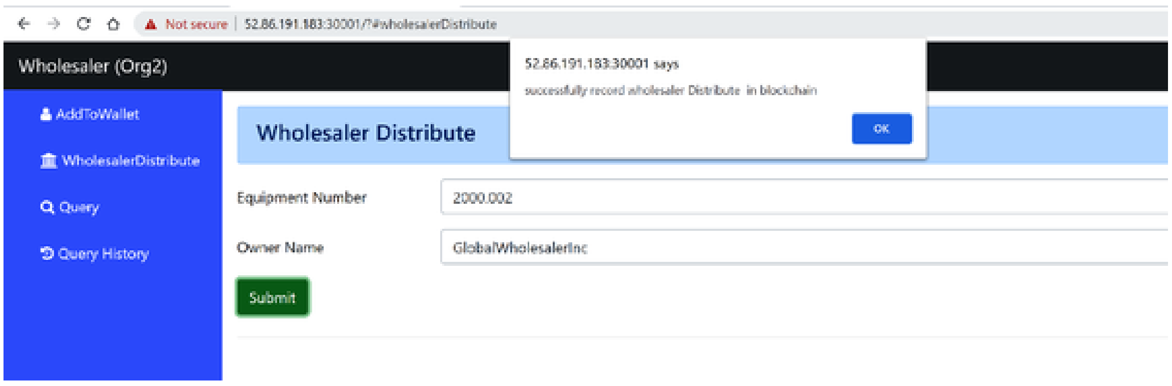
13. Make sure to update the base URL to *http://your-machine-public-ip:30001* in *plnClient.js*.

Then navigate back to the *pharma-ledger-network/organizations/wholesaler/application* folder and run the following:

```
npm install

node app.js
```

This starts the wholesaler web App. Open a browser and enter: http://*your-machine-public-ip*:30001


52.86.191.183:30001/?#wholesalerDistribute

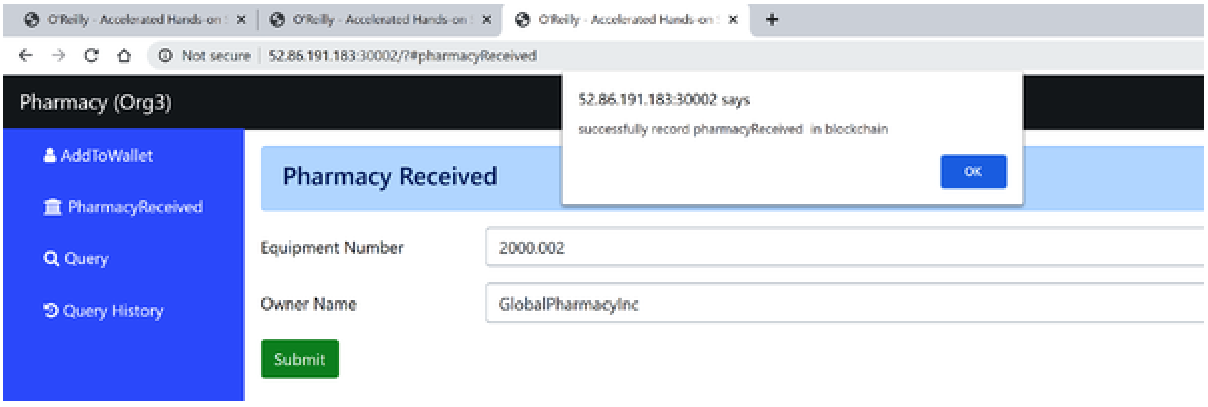
52.86.191.183:30001 says

successfully record wholesaler Distribute in blockchain

Figure 12: Adding a user (Alice) to the wholesaler

14. Add Alice as a wholesaler user (***Figure 12***) and submit a wholesalerDistribute request:

15. Follow the same steps by bringing up the pharmacy node server and add Eve as a pharmacy user (***Figure 13***). Submit a pharmacyReceived request:

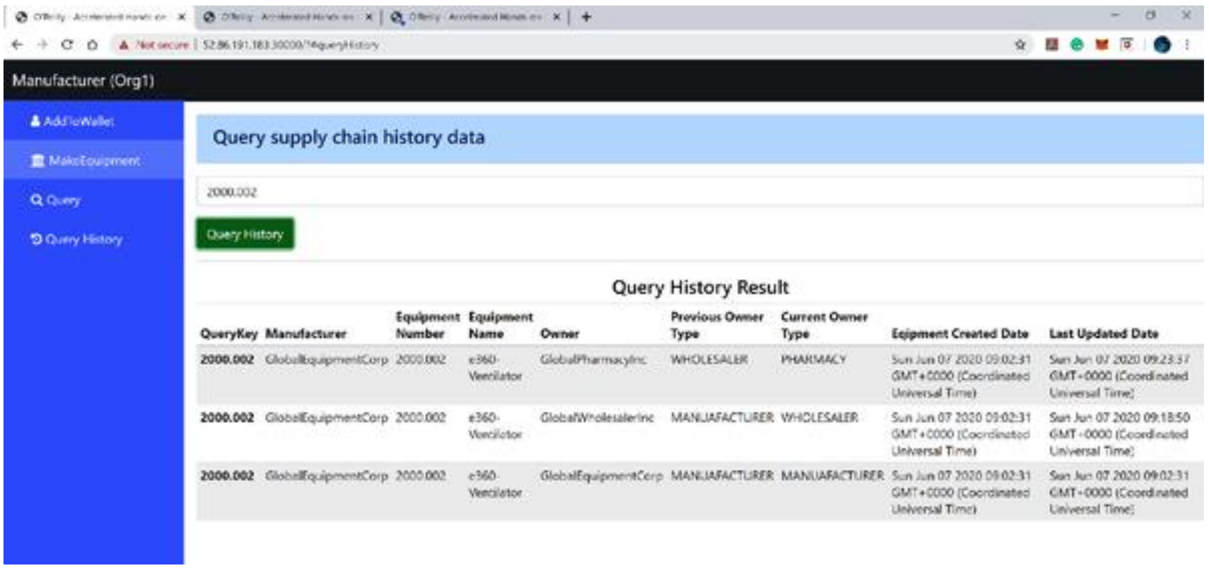

52.86.191.183:30002/?#pharmacyReceived

52.86.191.183:30002 says

successfully record pharmacyReceived in blockchain

Figure 13: Adding a user (Eve) to the pharmacy

Now the pharma ledger supply chain flow ends. Bob, Alice, and Eve can query equipment data and trace the entire supply chain process by querying historical data. Simply go to any user, and on the Query History page, search equipment **2000.002**, and you should see all query history results, as shown in ***Figure 14***.



Query History Result

QueryKey	Manufacturer	Equipment Number	Equipment Name	Owner	Previous Owner Type	Current Owner Type	Eqipment Created Date	Last Updated Date
2000.002	GlobalEquipmentCorp	2000.002	e360-Ventilator	GlobalPharmacyInc	WHOLESALER	PHARMACY	Sun Jun 07 2020 09:02:31 GMT+0000 (Coordinated Universal Time)	Sun Jun 07 2020 09:23:37 GMT+0000 (Coordinated Universal Time)
2000.002	GlobalEquipmentCorp	2000.002	e360-Ventilator	GlobalWholesalerInc	MANUAFACTURER	WHOLESALER	Sun Jun 07 2020 09:02:31 GMT+0000 (Coordinated Universal Time)	Sun Jun 07 2020 09:18:50 GMT+0000 (Coordinated Universal Time)
2000.002	GlobalEquipmentCorp	2000.002	e360-Ventilator	GlobalEquipmentCorp	MANUAFACTURER	MANUAFACTURER	Sun Jun 07 2020 09:02:31 GMT+0000 (Coordinated Universal Time)	Sun Jun 07 2020 09:02:31 GMT+0000 (Coordinated Universal Time)

Figure 14: Querying equipment historical data

Summary

In this session, you've gained valuable insights on building supply chain DApps using Hyperledger Fabric. Throughout the session, we covered various aspects of developing a consortium, analyzed the Pharma Ledger Network's life cycle, and traced equipment's transaction history.

We delved deep into writing chaincode for smart contracts, incorporating the logic for the manufacturer, wholesaler, and pharmacy. We set up the pharma ledger Fabric network environment and deployed the smart contract to the blockchain by following step-by-step instructions.

To ensure that all functions are functioning as expected, we tested a smart contract function using a command-line script. We then moved on to developing a UI page, where we demonstrated how to add users to a wallet and connect to the Fabric blockchain using SDK. We also created UI pages for the manufacturer, wholesaler, and pharmacy, enabling users from these organizations to invoke the smart contract by submitting related requests to the PLN blockchain.

By the end of the session, you have learned to build supply chain DApps using Hyperledger Fabric, write chaincode, test smart contracts, and develop user-friendly UI pages. With this knowledge, you'll be better equipped to develop robust and secure supply chain solutions.