☑ Complete Experiment: Implementing Private Data Collections in Hyperledger Fabric

🔧 STEP 1: Set up the Test Network

```
cd fabric-samples/test-network
./network.sh down  # (if already running)
./network.sh up createChannel -c mychannel -s couchdb
```

🗀 STEP 2: Create Chaincode Directory & Files

```
cd ..
mkdir asset-private-chaincode
cd asset-private-chaincode
```

🗎 File 1: package.json

```
{
  "name": "asset-private-chaincode",
  "version": "1.0.0",
  "main": "index.js",
  "dependencies": {
    "fabric-shim": "^2.4.0"
  }
}
```

Create this using:

```
nano package.json
```

Paste and save.

File 2: index.js

This is your actual **chaincode** file.

```
'use strict';

const { Contract } = require('fabric-shim');

class PrivateAssetContract extends Contract {
  async CreatePrivateAsset(ctx, assetID) {
    const transientData = ctx.stub.getTransient();
    if (!transientData.has('asset_properties')) {
      throw new Error('Missing transient data: asset_properties');
    }
    const asset = JSON.parse(transientData.get('asset_properties').toString());
    await ctx.stub.putPrivateData('collectionPrivateAssets', assetID,
Buffer.from(JSON.stringify(asset)));
  }

  async ReadPrivateAsset(ctx, assetID) {
    const assetBytes = await ctx.stub.getPrivateData('collectionPrivateAssets', assetID);
    if (!assetBytes || assetBytes.length === 0) {
      throw new Error(`Asset ${assetID} not found`);
    }
    return assetBytes.toString();
  }

  async DeletePrivateAsset(ctx, assetID) {
    await ctx.stub.deletePrivateData('collectionPrivateAssets', assetID);
  }
}

module.exports = PrivateAssetContract;
```

Create this using:

nano index.js

Paste and save.

📄 File 3: collections-config.json

```
[
  {
    "name": "collectionPrivateAssets",
    "policy": "OR('Org1MSP.member','Org2MSP.member')",
    "requiredPeerCount": 1,
    "maxPeerCount": 2,
    "blockToLive": 100,
    "memberOnlyRead": true
  }
]
```

Create this using:

nano collections-config.json

Paste and save.

📦 Install dependencies (optional for testing or later packaging)

npm install

🚀 STEP 3: Deploy Chaincode with PDC

Go back to test-network:

cd ../test-network

Deploy chaincode with private data collection config:

```
./network.sh deployCC \
  -ccn privatecc \
  -ccp ../asset-private-chaincode \
  -ccl javascript \
  -cccg ../asset-private-chaincode/collections-config.json
```

🚀 STEP 4: Invoke Chaincode Using Transient Data

Set environment variables (for Org1):

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
. ./scripts/envVar.sh
setGlobals 1
```

Invoke chaincode to create private asset:

```
peer chaincode invoke -o localhost:7050 \
--ordererTLSHostnameOverride orderer.example.com \
--tls --cafile "$ORDERER_CA" \
-C mychannel -n privatecc \
--peerAddresses localhost:7051 \
--tlsRootCertFiles "$PEER0_ORG1_CA" \
--transient "{\"asset_properties\":\"$(echo -n '{\"owner\":\"Bose\",\"value\":5000}' | base64)\"}" \
-c '{"function":"CreatePrivateAsset","Args":["asset1"]}'
```

☑️ Optional: Query Private Asset

```
peer chaincode query -C mychannel -n privatecc -c
'{"function":"ReadPrivateAsset","Args":["asset1"]}'
```

This will **only work** on a peer that is authorized (Org1 or Org2) in the collection policy.

🎯 Summary of File Names

File Name	Purpose
index.js	Main chaincode logic
package.json	Node.js metadata & dependencies
collections-config.json	Private data collection policy

Let me know if you want a simple Node.js **client app** to interact with this chaincode too!

To help you with your **practical exam tasks** on Hyperledger Fabric, I'll provide the code for **deploying and interacting with "HelloWorld" chaincode** and also **interacting with ERC20-like chaincode** in Hyperledger Fabric. Here's how you can approach both:

1. Deploy & Interact with HelloWorld Chaincode Using Hyperledger Fabric

Step 1: Set Up the Network

Before you deploy the chaincode, you need to start the **Fabric test network**. You can use the **Hyperledger Fabric samples** repository that comes with a predefined test network.

- Download the **Fabric samples** if you haven't already:

```
git clone https://github.com/hyperledger/fabric-samples.git
cd fabric-samples
```

- Start the test network:

```
cd first-network
./byfn.sh up
```

This will bring up a simple Hyperledger Fabric test network with two organizations and a channel.

Step 2: Write the "HelloWorld" Chaincode

Create a `helloworld` directory for your chaincode:

```
mkdir helloworld
cd helloworld
```

Create a file named `chaincode.js` with the following content:

```
'use strict';

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

class HelloWorldContract extends Contract {

    // The init method is called during the deployment of the
chaincode
    async initLedger(ctx) {
        console.log('HelloWorld contract is deployed!');
    }

    // This function returns a "Hello, World!" message
    async sayHello(ctx) {
        return 'Hello, World!';
    }
}

module.exports = HelloWorldContract;
```

This simple contract will have an `initLedger` function (called during the deployment of the chaincode) and a `sayHello` function that returns a "Hello, World!" message.

Step 3: Package and Deploy the Chaincode

Now that the chaincode is written, you need to deploy it to the network.

1. **Install the chaincode on the peers**:

```
peer chaincode install -n helloworld -v 1.0 -p ../helloworld
```

2. **Instantiate the chaincode** on the channel:

```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n
helloworld -v 1.0 -c '{"Args":[]}'
```

Step 4: Interact with the Chaincode

To interact with the deployed chaincode, use the `peer chaincode invoke` or `peer chaincode query` commands:

1. **Invoke the `sayHello` function**:

```
peer chaincode invoke -C mychannel -n helloworld -c
'{"Args":["sayHello"]}'
```

2. **Query the chaincode** to invoke the `sayHello` function:

```
peer chaincode query -C mychannel -n helloworld -c
'{"Args":["sayHello"]}'
```

This will return:

```
"Hello, World!"
```

2. Interacting with ERC20 Chaincode in Hyperledger Fabric

Step 1: Create ERC20 Chaincode

The ERC20 standard is used for creating tokens. Here's how you can implement an **ERC20-like chaincode** in Hyperledger Fabric.

Create a new file called `erc20.js`:

```
'use strict';

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

class ERC20Contract extends Contract {

    constructor() {
        super('org.example.token');
    }

    // Initializes the ledger with some tokens
    async initLedger(ctx) {
        console.log('Initializing ERC20 Ledger');
        const initialSupply = 1000000;
        const initialAccount =
'0x1234567890abcdef1234567890abcdef12345678';

        await ctx.stub.putState('totalSupply',
Buffer.from(initialSupply.toString()));
        await ctx.stub.putState(initialAccount,
Buffer.from(initialSupply.toString()));
    }

    // Returns the total supply of tokens
    async totalSupply(ctx) {
        const supply = await ctx.stub.getState('totalSupply');
        return supply.toString();
    }

    // Allows transferring tokens
    async transfer(ctx, from, to, amount) {
        const senderBalance = await ctx.stub.getState(from);
        if (!senderBalance || senderBalance.toString() === '0') {
            throw new Error('Insufficient balance');
        }

        const receiverBalance = await ctx.stub.getState(to);
        const newSenderBalance = parseInt(senderBalance.toString()) -
parseInt(amount);
```

```
        const newReceiverBalance = receiverBalance ?
parseInt(receiverBalance.toString()) + parseInt(amount) :
parseInt(amount);

        await ctx.stub.putState(from,
Buffer.from(newSenderBalance.toString()));
        await ctx.stub.putState(to,
Buffer.from(newReceiverBalance.toString()));

        return `${amount} tokens transferred from ${from} to ${to}`;
    }

    // Query balance of an account
    async balanceOf(ctx, account) {
        const balance = await ctx.stub.getState(account);
        return balance.toString();
    }
}

module.exports = ERC20Contract;
```

This `ERC20Contract` provides basic ERC20 functionality:

- `initLedger`: Initializes the token supply.
- `totalSupply`: Returns the total supply of tokens.
- `transfer`: Allows users to transfer tokens.
- `balanceOf`: Returns the balance of an account.

Step 2: Deploy the ERC20 Chaincode

1. **Install the ERC20 chaincode** on the peers:

```
peer chaincode install -n erc20 -v 1.0 -p ../erc20
```

2. **Instantiate the ERC20 chaincode** on the channel:

```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n
erc20 -v 1.0 -c '{"Args":[]}'
```

Step 3: Interact with the ERC20 Chaincode

1. **Query total supply**:

```
peer chaincode query -C mychannel -n erc20 -c
'{"Args":["totalSupply"]}'
```

2. **Query an account balance**:

```
peer chaincode query -C mychannel -n erc20 -c
'{"Args":["balanceOf","0x1234567890abcdef1234567890abcdef12345678"]}'
```

3. **Transfer tokens** from one account to another:

```
peer chaincode invoke -C mychannel -n erc20 -c
'{"Args":["transfer","0x1234567890abcdef1234567890abcdef12345678","0xa
bcdefabcdefabcdefabcdefabcdefabcdef","500"]}'
```

4. **Check the balance after transfer**:

```
peer chaincode query -C mychannel -n erc20 -c
'{"Args":["balanceOf","0x1234567890abcdef1234567890abcdef12345678"]}'
peer chaincode query -C mychannel -n erc20 -c
'{"Args":["balanceOf","0xabcdefabcdefabcdefabcdefabcdefabcdef"]}'
```

Summary of What Each Chaincode Does:

1. **HelloWorld Chaincode**: A simple contract with a `sayHello` function that returns a "Hello, World!" message. This is often used as a basic example to understand chaincode deployment and interaction.
2. **ERC20 Chaincode**: Implements basic ERC20 token functionality, including:
    a. Initializing token supply.

b. Querying the total supply.
c. Transferring tokens between accounts.
d. Querying the balance of an account.

Let me know if you need further explanations or help with running the commands!

Sure! Below are the code snippets for **developing a Message Storage module** and **implementing CRUD operations with Attribute-Based Data Access Control (ABAC)** in Hyperledger Fabric.

1. Develop and Interact with Message Storage Module using Hyperledger Fabric

This example demonstrates how to develop a simple message storage chaincode that allows users to store and retrieve messages in the Hyperledger Fabric ledger.

Step 1: Chaincode (MessageStorageChaincode.js)

```
'use strict';

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

class MessageStorageContract extends Contract {

    // Initialize the ledger with some messages (optional)
    async initLedger(ctx) {
        console.log('MessageStorage contract is deployed!');
    }

    // Store a message
    async storeMessage(ctx, messageId, messageContent) {
        console.log(`Storing message: ${messageId}`);
        const message = {
            content: messageContent,
            timestamp: new Date().toISOString(),
        };
        await ctx.stub.putState(messageId,
Buffer.from(JSON.stringify(message)));
        return JSON.stringify(message);
    }
```

```
    // Retrieve a message by ID
    async getMessage(ctx, messageId) {
        const messageAsBytes = await ctx.stub.getState(messageId);
        if (!messageAsBytes || messageAsBytes.length === 0) {
            throw new Error(`Message with ID ${messageId} does not
exist`);
        }
        return messageAsBytes.toString();
    }

    // Update an existing message
    async updateMessage(ctx, messageId, newContent) {
        const messageAsBytes = await ctx.stub.getState(messageId);
        if (!messageAsBytes || messageAsBytes.length === 0) {
            throw new Error(`Message with ID ${messageId} does not
exist`);
        }
        const message = JSON.parse(messageAsBytes.toString());
        message.content = newContent;
        message.timestamp = new Date().toISOString();
        await ctx.stub.putState(messageId,
Buffer.from(JSON.stringify(message)));
        return JSON.stringify(message);
    }

    // Delete a message
    async deleteMessage(ctx, messageId) {
        const messageAsBytes = await ctx.stub.getState(messageId);
        if (!messageAsBytes || messageAsBytes.length === 0) {
            throw new Error(`Message with ID ${messageId} does not
exist`);
        }
        await ctx.stub.deleteState(messageId);
        return `Message with ID ${messageId} deleted successfully`;
    }
}
```

```
module.exports = MessageStorageContract;
```

Step 2: Interacting with the Chaincode (Using Node.js)

Once the chaincode is deployed, you can interact with it by using the Fabric SDK in Node.js.

```
const { FileSystemWallet, Gateway } = require('fabric-network');
const path = require('path');

async function main() {
    const walletPath = path.join(__dirname, 'wallet');
    const wallet = new FileSystemWallet(walletPath);
    const gateway = new Gateway();

    await gateway.connect('connection-profile.json', {
        wallet,
        identity: 'user1',
        discovery: { enabled: true, asLocalhost: true },
    });

    const network = await gateway.getNetwork('mychannel');
    const contract = network.getContract('message-storage');

    // Store a message
    await contract.submitTransaction('storeMessage', 'msg1', 'Hello,
World!');
    console.log('Message stored successfully');

    // Retrieve the message
    const message = await contract.evaluateTransaction('getMessage',
'msg1');
    console.log('Retrieved message:', message.toString());

    // Update the message
    await contract.submitTransaction('updateMessage', 'msg1', 'Updated
message content');
    console.log('Message updated successfully');
```

```
    // Delete the message
    await contract.submitTransaction('deleteMessage', 'msg1');
    console.log('Message deleted successfully');

    await gateway.disconnect();
}

main().catch(console.error);
```

2. Implementing CRUD Operations with Attribute-Based Data Access Control (ABAC)

In this example, we'll use ABAC to control access to CRUD operations based on user attributes (e.g., role or department). The chaincode will check the attributes of the identity invoking the transaction before allowing access.

Step 1: Chaincode (ABACChaincode.js)

```
'use strict';

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

class ABACChaincode extends Contract {

    // Create an asset with ABAC check
    async createAsset(ctx, assetId, value) {
        const creator = ctx.clientIdentity.getID(); // Get the ID of
the caller
        const role = ctx.clientIdentity.getAttributeValue('role'); //
Get therole attribute

        // Only allow "admin" users to create assets
        if (role !== 'admin') {
            throw new Error(`Access denied: Only admins can create
assets`);
        }

        const asset = {
```

```
            value: value,
            createdBy: creator,
            timestamp: new Date().toISOString(),
        };

        await ctx.stub.putState(assetId,
Buffer.from(JSON.stringify(asset)));
        return JSON.stringify(asset);
    }

    // Read asset details
    async readAsset(ctx, assetId) {
        const assetAsBytes = await ctx.stub.getState(assetId);
        if (!assetAsBytes || assetAsBytes.length === 0) {
            throw new Error(`Asset with ID ${assetId} does not
exist`);
        }
        return assetAsBytes.toString();
    }

    // Update an asset with ABAC check
    async updateAsset(ctx, assetId, newValue) {
        const creator = ctx.clientIdentity.getID(); // Get the ID of
the caller
        const role = ctx.clientIdentity.getAttributeValue('role'); //
Get the role attribute

        // Only allow "admin" or "manager" roles to update assets
        if (role !== 'admin' && role !== 'manager') {
            throw new Error(`Access denied: Only admins or managers
can update assets`);
        }

        const assetAsBytes = await ctx.stub.getState(assetId);
        if (!assetAsBytes || assetAsBytes.length === 0) {
            throw new Error(`Asset with ID ${assetId} does not
exist`);
        }
```

```
        const asset = JSON.parse(assetAsBytes.toString());
        asset.value = newValue;
        asset.timestamp = new Date().toISOString();
        await ctx.stub.putState(assetId,
Buffer.from(JSON.stringify(asset)));
        return JSON.stringify(asset);
    }

    // Delete an asset with ABAC check
    async deleteAsset(ctx, assetId) {
        const creator = ctx.clientIdentity.getID(); // Get the ID of
the caller
        const role = ctx.clientIdentity.getAttributeValue('role'); //
Get the role attribute

        // Only allow "admin" users to delete assets
        if (role !== 'admin') {
            throw new Error(`Access denied: Only admins can delete
assets`);
        }

        const assetAsBytes = await ctx.stub.getState(assetId);
        if (!assetAsBytes || assetAsBytes.length === 0) {
            throw new Error(`Asset with ID ${assetId} does not
exist`);
        }

        await ctx.stub.deleteState(assetId);
        return `Asset with ID ${assetId} deleted successfully`;
    }
}

module.exports = ABACChaincode;
```

Step 2: Interacting with the Chaincode (Using Node.js)

```
const { FileSystemWallet, Gateway } = require('fabric-network');
const path = require('path');
```

```
async function main() {
    const walletPath = path.join(__dirname, 'wallet');
    const wallet = new FileSystemWallet(walletPath);
    const gateway = new Gateway();

    await gateway.connect('connection-profile.json', {
        wallet,
        identity: 'user1',
        discovery: { enabled: true, asLocalhost: true },
    });

    const network = await gateway.getNetwork('mychannel');
    const contract = network.getContract('abac-chaincode');

    // Create an asset (only "admin" can do this)
    await contract.submitTransaction('createAsset', 'asset1', '1000');
    console.log('Asset created successfully');

    // Read the asset
    const asset = await contract.evaluateTransaction('readAsset',
'asset1');
    console.log('Asset details:', asset.toString());

    // Update the asset (only "admin" or "manager" can do this)
    await contract.submitTransaction('updateAsset', 'asset1', '2000');
    console.log('Asset updated successfully');

    // Delete the asset (only "admin" can do this)
    await contract.submitTransaction('deleteAsset', 'asset1');
    console.log('Asset deleted successfully');

    await gateway.disconnect();
}

main().catch(console.error);
```

Explanation of ABAC Example:

- **ABAC (Attribute-Based Access Control)**: Access control is based on the attributes of the user invoking the transaction (e.g., role).
- **Role-based access**: The role of the user determines whether they can create, update, or delete assets. For example, only "admin" users can create or delete assets, while both "admin" and "manager" users can update assets.

Conclusion:

These examples cover both the **message storage module** and **ABAC-based CRUD operations** in Hyperledger Fabric. You can extend these examples based on your specific business logic and use cases. Let me know if you need further explanations or have additional questions!

Here's how you can approach both experiments in **Hyperledger Fabric using Node.js chaincode** with clarity and sample code snippets:

🔐 1. Deploy & Interact with Encryption Module using Hyperledger Fabric

📌 Goal:

Store encrypted data on the ledger and decrypt it during retrieval, ensuring confidentiality.

🧩 Chaincode (`encryption-contract.js`)

```
'use strict';

const { Contract } = require('fabric-contract-api');
const crypto = require('crypto');

class EncryptionContract extends Contract {
    async initLedger(ctx) {
        console.log('EncryptionContract deployed.');
    }

    async storeEncrypted(ctx, key, plainText) {
```

```
        const cipher = crypto.createCipher('aes-256-cbc', 'secret-
key');
        let encrypted = cipher.update(plainText, 'utf8', 'hex');
        encrypted += cipher.final('hex');

        await ctx.stub.putState(key, Buffer.from(encrypted));
        return `Data stored encrypted under key ${key}`;
    }

    async retrieveDecrypted(ctx, key) {
        const encrypted = await ctx.stub.getState(key);
        if (!encrypted || encrypted.length === 0) {
            throw new Error(`No data found for key ${key}`);
        }

        const decipher = crypto.createDecipher('aes-256-cbc', 'secret-
key');
        let decrypted = decipher.update(encrypted.toString(), 'hex',
'utf8');
        decrypted += decipher.final('utf8');

        return decrypted;
    }
}

module.exports = EncryptionContract;
```

📄 2. Working with Secure Agreements in Hyperledger Fabric

📌 Goal:

Create a tamper-proof agreement between two parties, and ensure both parties' signatures are recorded.

Chaincode (agreement-contract.js)

```
'use strict';
const { Contract } = require('fabric-contract-api');

class AgreementContract extends Contract {
    async initLedger(ctx) {
        console.log('AgreementContract deployed.');
    }

    async createAgreement(ctx, agreementId, party1, party2, content) {
        const agreement = {
            agreementId,
            party1,
            party2,
            content,
            signedBy: [],
            timestamp: new Date().toISOString()
        };
        await ctx.stub.putState(agreementId,
Buffer.from(JSON.stringify(agreement)));
        return `Agreement ${agreementId} created`;
    }

    async signAgreement(ctx, agreementId, party) {
        const data = await ctx.stub.getState(agreementId);
        if (!data || data.length === 0) {
            throw new Error(`Agreement ${agreementId} not found`);
        }

        const agreement = JSON.parse(data.toString());

        if (!agreement.signedBy.includes(party)) {
            agreement.signedBy.push(party);
        }

        await ctx.stub.putState(agreementId,
Buffer.from(JSON.stringify(agreement)));
        return `Agreement ${agreementId} signed by ${party}`;
```

```
    }

    async readAgreement(ctx, agreementId) {
        const data = await ctx.stub.getState(agreementId);
        if (!data || data.length === 0) {
            throw new Error(`Agreement ${agreementId} not found`);
        }

        return data.toString();
    }
}

module.exports = AgreementContract;
```

🚀 How to Deploy

Using `fabric-samples/test-network`:

```
./network.sh up createChannel
./network.sh deployCC -ccn encryptioncc -ccp ../encryption-chaincode -
ccl javascript
./network.sh deployCC -ccn agreementcc -ccp ../agreement-chaincode -
ccl javascript
```

Replace `-ccp` with your actual path to the chaincode directory.

☑ For Practical Exams:

Write the following parts:

- `initLedger`, one store/retrieve method for encryption.
- One `createAgreement` and `signAgreement` method.
- CLI or Node.js script to invoke and query.

Let me know if you want client-side (Node.js) scripts to interact with these chaincodes!

Here's a clear breakdown of the **projects/experiments** you're asking about — including **Hyperledger Fabric** chaincodes and a high-level overview for the **Ethereum/Polygon bridging** and **DID management**.

☑ 1. Basic Asset Transfer Chaincode using Hyperledger Fabric

🧩 Chaincode (`assetTransfer.js`)

```
'use strict';

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

class AssetTransfer extends Contract {
    async initLedger(ctx) {
        const assets = [
            { ID: 'asset1', Color: 'blue', Size: 5, Owner: 'Tom',
AppraisedValue: 300 },
            { ID: 'asset2', Color: 'red', Size: 6, Owner: 'Jerry',
AppraisedValue: 400 }
        ];
        for (const asset of assets) {
            await ctx.stub.putState(asset.ID,
Buffer.from(JSON.stringify(asset)));
        }
    }

    async createAsset(ctx, id, color, size, owner, value) {
        const asset = { ID: id, Color: color, Size: parseInt(size),
Owner: owner, AppraisedValue: parseInt(value) };
        await ctx.stub.putState(id,
Buffer.from(JSON.stringify(asset)));
    }

    async readAsset(ctx, id) {
```

```
        const data = await ctx.stub.getState(id);
        if (!data || data.length === 0) throw new Error(`Asset ${id}
not found`);
        return data.toString();
    }

    async transferAsset(ctx, id, newOwner) {
        const assetJSON = await ctx.stub.getState(id);
        if (!assetJSON || assetJSON.length === 0) throw new
Error(`Asset ${id} not found`);
        const asset = JSON.parse(assetJSON.toString());
        asset.Owner = newOwner;
        await ctx.stub.putState(id,
Buffer.from(JSON.stringify(asset)));
    }
}

module.exports = AssetTransfer;
```

🔗 2. Token Bridging Between Ethereum and Polygon

🧠 Concept:

This involves **moving tokens** from Ethereum to Polygon via a **bridge contract**. Usually, this is done by:

- Locking the token on Ethereum
- Minting an equivalent token on Polygon

🛠️ Tools:

- Use Hardhat or Truffle for deployment
- Deploy ERC-20 token + Bridge contract on Ethereum
- Deploy MirrorToken + Bridge contract on Polygon

📝 You don't write a chaincode here — it's done in Solidity.

Do you want **Solidity sample code** for this?

🪪 3. Implementation of Decentralized Identity (DID) in Hyperledger Fabric

🧩 Approach:

1. Issue DIDs to users using Fabric.
2. Store identity metadata on ledger.
3. Only allow updates by identity owners.

🧩 Chaincode snippet:

```
'use strict';

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

class DIDContract extends Contract {
    async registerDID(ctx, did, publicKey) {
        const identity = { did, publicKey, controller:
ctx.clientIdentity.getID() };
        await ctx.stub.putState(did,
Buffer.from(JSON.stringify(identity)));
    }

    async resolveDID(ctx, did) {
        const result = await ctx.stub.getState(did);
        if (!result || result.length === 0) throw new Error('DID not
found');
        return result.toString();
    }
}

module.exports = DIDContract;
```

🚚 4. Supply Chain Management Chaincode

🧩 Chaincode (`supplyChain.js`)

```
'use strict';

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

class SupplyChain extends Contract {
    async createItem(ctx, id, name, origin) {
        const item = {
            id,
            name,
            origin,
            status: 'Created',
            currentOwner: origin
        };
        await ctx.stub.putState(id,
Buffer.from(JSON.stringify(item)));
    }

    async updateStatus(ctx, id, status, owner) {
        const data = await ctx.stub.getState(id);
        if (!data || data.length === 0) throw new Error(`Item ${id}
not found`);
        const item = JSON.parse(data.toString());
        item.status = status;
        item.currentOwner = owner;
        await ctx.stub.putState(id,
Buffer.from(JSON.stringify(item)));
    }

    async getItem(ctx, id) {
        const data = await ctx.stub.getState(id);
        if (!data || data.length === 0) throw new Error(`Item ${id}
not found`);
        return data.toString();
    }
```

```
}

module.exports = SupplyChain;
```

🚀 How to Deploy (for all Hyperledger projects)

From inside the `fabric-samples/test-network`:

```
./network.sh up createChannel
./network.sh deployCC -ccn <cc-name> -ccp ../path-to-chaincode -ccl
javascript
```

Would you like:

- Node.js client app to test these?
- Sample `invoke` commands?
- Full directory structure?

Let me know which one you want detailed next!

Great! Let's walk through a **simple Ethereum to Polygon token bridge**. This process allows tokens to be **moved (bridged)** from Ethereum to Polygon.

🔁 Token Bridge from Ethereum to Polygon – Overview

You'll deploy:

◇ On Ethereum:

- A **Token contract** (ERC20)
- A **Bridge contract** to lock tokens

🔸 On Polygon:

- A **MirrorToken contract** (replica of ERC20)
- A **Bridge contract** to mint/burn tokens

✅ This is a *basic simulation* (not production-level with validators or oracles like Polygon PoS bridge). But it shows the working idea.

🧱 1. ERC20 Token Contract (on Ethereum)

```
// contracts/MyToken.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("BridgeToken", "BRG") {
        _mint(msg.sender, initialSupply);
    }
}
```

🔐 2. Ethereum Bridge Contract

```
// contracts/EthBridge.sol
pragma solidity ^0.8.0;

interface IToken {
    function transferFrom(address, address, uint256) external returns
(bool);
}

contract EthBridge {
    address public admin;
```

```
    address public token;

    event Locked(address indexed user, uint256 amount);

    constructor(address _token) {
        admin = msg.sender;
        token = _token;
    }

    function lockTokens(uint256 amount) external {
        IToken(token).transferFrom(msg.sender, address(this), amount);
        emit Locked(msg.sender, amount); // Simulated proof to trigger
Polygon minting
    }
}
```

3. Mirror Token on Polygon

```
// contracts/MirrorToken.sol
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MirrorToken is ERC20 {
    address public bridge;

    constructor() ERC20("BridgeToken", "BRG") {
        bridge = msg.sender; // Only bridge can mint
    }

    function mint(address to, uint256 amount) external {
        require(msg.sender == bridge, "Only bridge can mint");
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) external {
```

```
        require(msg.sender == bridge, "Only bridge can burn");
        _burn(from, amount);
    }
}
```

🔗 4. Polygon Bridge Contract

```
// contracts/PolygonBridge.sol
pragma solidity ^0.8.0;

interface IMirrorToken {
    function mint(address, uint256) external;
    function burn(address, uint256) external;
}

contract PolygonBridge {
    address public token;
    address public admin;

    event Unlocked(address indexed user, uint256 amount);

    constructor(address _token) {
        admin = msg.sender;
        token = _token;
    }

    function mintOnPolygon(address to, uint256 amount) external {
        require(msg.sender == admin, "Only admin can mint"); //
Simulate oracle
        IMirrorToken(token).mint(to, amount);
    }

    function burnFromPolygon(address from, uint256 amount) external {
        require(msg.sender == from, "User must call");
        IMirrorToken(token).burn(from, amount);
        emit Unlocked(from, amount); // Simulated proof to unlock on
```

```
Ethereum
    }
}
```

⚙️ 5. How It Works (Flow):

→ Ethereum to Polygon

1. User approves & calls `EthBridge.lockTokens(amount)`
2. `Locked` event emitted
3. Admin/oracle sees event → calls `PolygonBridge.mintOnPolygon(user, amount)`

← Polygon to Ethereum

1. User calls `PolygonBridge.burnFromPolygon(amount)`
2. `Unlocked` event emitted
3. Admin/oracle sees event → manually sends tokens back on Ethereum if needed

🛠️ 6. Tools & Deployment

Use **Hardhat** to deploy:

```
npx hardhat compile
npx hardhat run scripts/deploy.js --network goerli # Ethereum
npx hardhat run scripts/deploy.js --network polygonMumbai # Polygon
```

Would you like:

- 🧪 Sample `deploy.js` scripts for Hardhat?
- 🖥️ Web frontend or script to test the bridging?
- 🛰️ Setup for cross-chain communication (simulated oracle)?

Let me know what level you need next!