To Join Diploma Wallah Group contact :- 9508550281

BLOCK CHAIN TECHNOLOGY

DIPLOMA WALLAH

OPEN ELECTIVE

Jharkhand University Of Technology (JUT)

UNIT - III: HYPERLEDGER FABRIC ARCHITECTURE AND DEVELOPMENT

Decomposing the Consensus Process

Consensus in blockchain systems can be decomposed into distinct phases and components to better understand how distributed nodes reach agreement. Unlike monolithic consensus mechanisms (like PoW), modern enterprise blockchains separate consensus into multiple specialized stages.geeksforgeeks+2

General Consensus Process Decomposition

Phase 1: Transaction Proposal

A participant (node) or leader proposes a new transaction or block that must comply with the network's protocol rules.geeksforgeeks+1

Phase 2: Transaction Broadcast

The proposed transaction or block is broadcast to other nodes in the network for validation.geeksforgeeks+1

Phase 3: Validation

Nodes verify the transaction or block based on predefined rules, checking signatures, permissions, and business logic.geeksforgeeks+1

Phase 4: Consensus Voting

Nodes communicate with each other to reach agreement on the proposed transaction or block. The specific voting method varies by consensus mechanism.persistent+2

Phase 5: Agreement and Finalization

Once a sufficient number of nodes agree on the proposed transaction or block, it is added to the blockchain.geeksforgeeks+1

Phase 6: Propagation

The updated ledger, now including the new block, is propagated to all nodes in the network to ensure they all have the same view of the blockchain.geeksforgeeks+1

Components of Consensus Mechanisms

Diplomawallah.in

Website:- Diplomawallah.in

Consensus protocols can be decomposed into several distinct components, each responsible for a specific aspect:venom

1. Election Component

- **Purpose:** Select a leader or set of validators responsible for proposing blocks.geeksforgeeks+1
- **Methods:**
    - Round-robin rotation
    - Stake-based selection
    - Random selection with cryptographic guarantees
    - Voting-based electiongeeksforgeeks+1

Example: In Raft, nodes elect a leader through randomized timeouts and majority voting. In Kafka-based ordering, leaders are elected through ZooKeeper coordination.geeksforgeeks

2. Proposal and Acceptance Component

- **Purpose:** Leader proposes a value or block; other nodes evaluate and accept or reject.venom+1
- **Process:**
    - Leader constructs a proposal containing transactions
    - Proposal broadcast to validator nodes
    - Validators check proposal validity
    - Validators signal acceptance or rejectiongeeksforgeeks

3. Agreement Component

- **Purpose:** Reach majority consensus on the proposed value.venom+1
- **Threshold:** Different mechanisms require different agreement thresholds:
    - Simple majority (>50%)
    - Supermajority (>66.6% for Byzantine Fault Tolerance)
    - Unanimous agreement (some permissioned systems)geeksforgeeks

4. Commitment Component

- **Purpose:** Finalize the agreed-upon decision and commit to the ledger.venom+1

Diplomawallah.in

- **Action:** Once consensus is reached, all nodes commit the agreed-upon values to their local copies of the blockchain or database.geeksforgeeks

Execute-Order-Validate vs. Order-Execute

Traditional Order-Execute Model (Bitcoin, Ethereum):

1. **Order:** Transactions are ordered into blocks by miners/validators
2. **Execute:** All nodes execute all transactions sequentially
3. **Validate:** Nodes validate execution results and update state

Limitations:

- Sequential execution limits throughput
- All nodes must execute all transactions
- Non-deterministic smart contracts can cause forks
- Limited privacy (all nodes see all data)educative+1

Hyperledger Fabric's Execute-Order-Validate Model:

1. **Execute (Endorsement):** Selected peers simulate transactions in parallel before ordering
2. **Order:** Ordering service sequences endorsed transactions into blocks
3. **Validate:** All peers validate endorsements and apply state changes

Advantages:

- Parallel transaction execution
- Privacy through selective endorsement
- Deterministic validation
- Better scalabilitygeeksforgeeks+1

This decomposition enables Hyperledger Fabric to achieve higher performance and flexibility compared to traditional blockchain architectures.

Hyperledger Fabric Components

Hyperledger Fabric is a modular, permissioned blockchain platform designed for enterprise use. Its architecture consists of several specialized components working together to provide a secure, scalable distributed ledger system.ibm+3

1. Peer Nodes

Definition: Peers are fundamental nodes in the Fabric network that host copies of ledgers and smart contracts (chaincode).softwaremill+3

Key functions:

- Maintain blockchain ledgers and world state databases
- Execute and host chaincode (smart contracts)
- Validate transactions
- Commit blocks to the ledger
- Communicate with other peers via gossip protocolhyperledger-fabric.readthedocs+2

Peer characteristics:

- A peer belongs to **one organization** but can participate in **multiple channels**softwaremill+1
- Each peer can host **multiple ledgers** (one per channel) and **multiple chaincodes**hyperledger-fabric.readthedocs+1
- Typically, organizations run multiple peers for redundancy and load handlingspydra+1

Types of peers:

a) Endorsing Peers

- Execute chaincode to simulate transactions
- Generate transaction proposals with read-write sets
- Sign endorsement responses
- Must satisfy endorsement policies for transactions to be valideducative+2

b) Committing Peers

- All peers are committing peers
- Receive blocks from orderers
- Validate transactions against endorsement policies
- Commit valid transactions to the ledger
- Update world state databasegeeksforgeeks+1

c) Anchor Peers

- Designated peers that other organizations can discover
- Facilitate cross-organization communication

Diplomawallah.in

- One per organization per channelsoftwaremill+1

d) Leader Peers

- Responsible for distributing transactions from orderers to other peers in the organization
- Elected dynamically or statically configuredhyperledger-fabric.readthedocs+1

Peer architecture components:

- **Ledger:** Stores blockchain (transaction log) and world state (current key-value pairs)
- **Chaincode containers:** Separate Docker containers running chaincode logic
- **Gossip protocol:** Peer-to-peer communication for data dissemination
- **State database:** LevelDB (default, embedded) or CouchDB (optional, external)stackoverflow+2

2. Orderer Nodes (Ordering Service)

Definition: Orderers are specialized nodes responsible for ordering transactions and creating blocks, but they do **not** execute transactions or maintain ledgers.geeksforgeeks+3

Primary functions:

a) Transaction Ordering

- Receive endorsed transactions from clients
- Order transactions chronologically across the network
- Package ordered transactions into blocks
- Distribute blocks to all committing peersibm+2

Critical distinction: If transactions were ordered at peers themselves, it would increase the possibility of state forks where different peers have different transaction orders.ibm

b) Maintaining Orderer System Channel

- Stores the **consortium** definition (list of organizations permitted to create channels)
- Maintains network-level configurationibm

c) Identity Validation

- Validates organization membership against consortium
- Enforces channel permissions

Diplomawallah.in

- Validates transaction channel configurationsibm

d) Channel Management

- Creates new channels
- Manages channel membership
- Enforces channel policiesgeeksforgeeks+1

Types of ordering services:

1. Solo Orderer

- Single orderer node
- No fault tolerance
- **Use case:** Development and testing onlygeeksforgeeks

2. Kafka Orderer (Deprecated)

- Cluster of orderers using Apache Kafka for consensus
- Crash fault tolerant (CFT)
- **Use case:** Production networks (older versions)geeksforgeeks

3. Raft Orderer (Current Recommended)

- Cluster of orderers using Raft consensus algorithm
- Crash fault tolerant (CFT)
- Simpler than Kafka, no external dependencies
- **Use case:** Production networks (Fabric 1.4.1+)geeksforgeeks+1

4. BFT Orderer (Future)

- Byzantine Fault Tolerant ordering
- Can tolerate malicious orderers
- Under developmentgeeksforgeeks

Orderer characteristics:

- Does **not** process smart contracts or execute transactionslinkedin+1
- Does **not** maintain ledgers (only orders and distributes)ibm
- Modular design allows pluggable consensus implementationslinkedin
- Can be shared across multiple channelsibm

3. Chaincode (Smart Contracts)

Diplomawallah.in

Definition: Chaincode is Hyperledger Fabric's term for smart contracts—programmable business logic that defines assets and transaction rules.aws.amazon+2

Supported languages:

- **Go (Golang):** Most mature and performant
- **Node.js (JavaScript/TypeScript):** Good developer experience
- **Java:** Enterprise-friendly optionaws.amazon+3

Chaincode lifecycle (Fabric 2.x):

1. **Package:** Bundle chaincode source code into deployable package
2. **Install:** Install chaincode package on endorsing peers
3. **Approve:** Organizations approve chaincode definition
4. **Commit:** Commit chaincode definition to channel once sufficient approvals received
5. **Invoke:** Execute chaincode functions via transactionshyperledger-fabric.readthedocs+1

Chaincode structure:

Basic components:

- **Init function:** Initializes chaincode state (optional in Fabric 2.x)
- **Invoke functions:** Business logic functions for querying and updating state
- **Shim library:** Provides APIs to interact with Fabric peersgithub+2

Example (Node.js):

javascript

```
const shim = require('fabric-shim');


class MyChaincode {
  async Init(stub) {
    // Initialization logic
    return shim.success();
  }
```

Diplomawallah.in

```
    async Invoke(stub) {
      let ret = stub.getFunctionAndParameters();
      let method = this[ret.fcn];
      return method(stub, ret.params);
    }

    async createAsset(stub, args) {
      // Business logic to create asset
      await stub.putState(key, value);
      return shim.success();
    }

    async queryAsset(stub, args) {
      // Business logic to query asset
      let value = await stub.getState(key);
      return shim.success(value);
    }
}

shim.start(new MyChaincode());
```

Chaincode execution model:

- Chaincode runs in **separate Docker containers**, isolated from peer processeshyperledger.github+1
- Executes in response to transaction proposals
- Reads from and writes to world state via CouchDB or LevelDB
- Cannot directly access external systems (must use oracles)aws.amazon+1

Chaincode APIs:

- putState(key, value): Write key-value pair to world state
- getState(key): Read value for key from world state

Diplomawallah.in

- delState(key): Delete key from world state
- getStateByRange(startKey, endKey): Range query
- getQueryResult(query): Rich query (CouchDB only)
- getHistoryForKey(key): Query all historical values for a keyhyperledger.github+2

4. Membership Service Provider (MSP)

Definition: MSP is the component that defines the rules by which identities are validated, authenticated, and given access to participate in a Hyperledger Fabric network.astconsulting+3

Primary functions:

a) Identity Management

- Defines which Certificate Authorities (CAs) are trusted
- Establishes organizational membership
- Maps certificates to roles and permissionsdev+2

b) Authentication

- Verifies participant identities using X.509 certificates
- Ensures only authorized entities can submit transactionsastconsulting+1

c) Authorization

- Enforces access control policies
- Determines what actions each identity can performdev+1

MSP structure:

Root Certificates:

- Root CA certificate: Establishes root of trust for organization
- TLS CA certificate: For secure peer-to-peer communicationibm+1

Intermediate Certificates (Optional):

- Intermediate CA certificates: Chain of trust between root and leaf certificatesibm

Admin Certificates:

- Identify administrators with elevated privileges
- Required for network operations (creating channels, installing chaincode)astconsulting+1

Diplomawallah.in

Node Certificates:

- Each peer, orderer, and client has unique identity certificate
- Organizational Unit (OU) attributes distinguish node typesibm

Revocation Lists (Optional):

- List of revoked certificates that should no longer be trustedhyperledger-fabric.readthedocs+1

Types of MSPs:

1. Local MSP

- Defined for each node (peer, orderer) and client
- Contains node's own identity certificates
- Controls what the node can dostackoverflow+2

2. Channel MSP

- Defined at channel level
- Lists all organizations participating in the channel
- Specifies which identities are members of each organization
- Enforced by all channel membersstackoverflow+2

MSP configuration:

Each MSP includes:

- MSP identifier (unique name)
- Root CA certificates (trust anchors)
- Admin certificates
- Organizational Units (optional)
- Certificate revocation lists (optional)hyperledger-fabric.readthedocs+1

Certificate Authority (CA) relationship:

MSPs work with CAs to manage the identity lifecycle:

1. **CA enrolls** participants and issues certificates
2. **MSP validates** certificates against trusted CA roots
3. **MSP authorizes** actions based on certificate attributesdev+2

Example: A university blockchain network might have:

Diplomawallah.in

- University MSP: Defines university administrators and professors (can create credentials)
- Student MSP: Defines students (can only read credentials)dev

5. Channels

Definition: Channels are private communication pathways that allow subsets of network participants to conduct confidential transactions.hyperledger-fabric.readthedocs+3

Key characteristics:

a) Data Isolation

- Each channel has a **completely separate ledger**hyperledger-fabric.readthedocs+1
- Transactions on one channel are invisible to other channelshyperledgendary.github+1
- Organizations can participate in multiple channels simultaneouslysoftwaremill+1

b) Privacy

- Only channel members can access channel data
- Different endorsement policies per channel
- Enables competing organizations to use same network without sharing sensitive datahyperledgendary.github

Diplomawallah.in

Channel components:

- **Channel configuration:** Defines members, policies, and parameters
- **Channel ledger:** Independent blockchain and world state
- **Channel MSPs:** Identity definitions for channel members
- **Chaincode:** Can be instantiated on multiple channels independentlyspydra+2

Use cases:

- Industry consortiums with competing members
- Supply chains with confidential pricing
- Financial networks with regulatory partitionshyperledgendary.github

6. Ledger

Definition: The ledger in Hyperledger Fabric consists of two components: the blockchain (transaction log) and the world state (current state database).hyperledger-fabric.readthedocs+3

Component 1: Blockchain (Transaction Log)

Structure:

- Immutable, sequenced record of all transactions
- Stored as a file-based ledger with serialized blocks
- Each block contains one or more transactions
- Blocks linked cryptographically via hasheshyperledger-fabric.readthedocs+2

Block structure:

- **Block header:** Block number, current block hash, previous block hash, transaction count
- **Block data:** Array of transactions with read-write sets
- **Block metadata:** Signatures, endorsements, validation codeshyperledger-fabric.readthedocs+1

Characteristics:

- Append-only (cannot modify or delete past transactions)
- Definitive source of truth
- Can be used to rebuild world state if neededstackoverflow+1

Diplomawallah.in

Component 2: World State (State Database)

Purpose:

- Holds **current values** for all keys ever included in the blockchaingithub+3
- Provides fast access to latest state without traversing entire blockchainhyperledger-fabric.readthedocs+1
- Automatically regenerated from blockchain if neededstackoverflow+1

State database options:

a) LevelDB (Default)

- Embedded key-value database
- Runs within peer process
- Supports key-based lookups and range queries
- Stores data as simple key-value pairsgithub+2

b) CouchDB (Optional)

- External NoSQL document database
- Stores data as JSON documents
- Supports rich queries using Mango query language
- Enables complex queries on chaincode data structure
- Better for applications requiring sophisticated queriesgithub+2

World state operations:

- **Create:** Add new key-value pair
- **Update:** Modify existing key's value
- **Delete:** Remove key-value pair
- **Query:** Retrieve current value(s)hyperledger-fabric.readthedocs+1

Ledger per channel:

- Each channel has its own independent ledger
- Peers maintain separate ledgers for each channel they join
- Prevents data leakage between channelshyperledger-fabric.readthedocs+2

Ledger features:

- Query using key-based lookups
- Range queries
- Composite key queries
- Rich queries (CouchDB only)
- Full history retrieval for any keyhyperledger-fabric.readthedocs+1

7. Certificate Authority (CA)

Definition: The CA issues and manages digital certificates for network participants.astconsulting+2

Functions:

- **Enrollment:** Issue identity certificates to new members
- **Re-enrollment:** Refresh certificates before expiration
- **Revocation:** Invalidate compromised certificates
- **Attribute management:** Assign roles and attributes to identitiesastconsulting+1

Diplomawallah.in

Fabric CA features:

- Native Fabric CA server (optional)
- Support for external CAs (enterprise PKI systems)
- Hierarchical CA structure (root and intermediate CAs)
- Dynamic enrollment and revocationibm+2

Additional Components:

8. Client Applications

- End-user interfaces that interact with the blockchain
- Use Fabric SDKs to submit transactions and query ledgers
- Run outside the blockchain networkdev+1

9. Fabric Gateway

- Simplifies client application development
- Routes transaction proposals to appropriate peers
- Handles endorsement collection
- Manages transaction submission to orderershyperledger-fabric.readthedocs

---

Chaincode Design and Implementation

Chaincode is the core programmable component of Hyperledger Fabric, defining business logic and rules for asset management. Proper chaincode design is critical for network performance, security, and maintainability.hyperledger-fabric.readthedocs+1

Chaincode Design Principles

1. Determinism

Requirement: Chaincode must produce the same output given the same input across all endorsing peers.aws.amazon+1

Implications:

- Avoid random number generation
- Don't use system timestamps (use transaction timestamps from stub)
- No external API calls during execution
- No file system accessaws.amazon

Diplomawallah.in

Why: Non-deterministic chaincode causes endorsement mismatches, making transactions invalid.

2. Statelessness

Principle: Chaincode should not maintain in-memory state between invocations.hyperledger-fabric.readthedocs+1

Reason: Each chaincode invocation runs in isolated container; state must be persisted to ledger.

Best practice: Always read from and write to world state via stub APIs.

3. Modularity

Design: Separate business logic into distinct functions.hyperledger-fabric.readthedocs+1

Benefits:

- Easier testing and maintenance
- Clear endorsement policy mapping
- Reusable components

4. Error Handling

Requirement: Return appropriate success or error responses.hyperledger.github+2

Best practices:

- Validate all inputs
- Check for null/undefined values
- Return descriptive error messages
- Use shim.success() and shim.error() appropriatelyhyperledger.github

5. Security

Considerations:

- Validate caller identity
- Enforce access controls
- Sanitize inputs to prevent injection attacks
- Protect sensitive dataaws.amazon

Chaincode Structure and Components

Basic Chaincode Template (Node.js):

Diplomawallah.in

javascript

```
const shim = require('fabric-shim');
const util = require('util');

class MyChaincode {

  // Initialize chaincode (optional in Fabric 2.x)
  async Init(stub) {
    console.info('========= MyChaincode Init =========');
    // Initialize ledger with default data if needed
    await stub.putState('initialized', Buffer.from('true'));
    return shim.success();
  }

  // Main entry point for invocations
  async Invoke(stub) {
    console.info('========= MyChaincode Invoke =========');

    // Get function name and parameters
    let ret = stub.getFunctionAndParameters();
    console.info('Function: ' + ret.fcn);
    console.info('Parameters: ' + ret.params);

    // Route to appropriate function
    let method = this[ret.fcn];
    if (!method) {
      return shim.error('Function ' + ret.fcn + ' not found');
    }
```

Diplomawallah.in

```
        try {
            let response = await method(stub, ret.params);
            return shim.success(response);
        } catch (err) {
            return shim.error(err.message);
        }
    }

    // Business logic functions
    async createAsset(stub, args) {
        // Implementation
    }

    async readAsset(stub, args) {
        // Implementation
    }

    async updateAsset(stub, args) {
        // Implementation
    }

    async deleteAsset(stub, args) {
        // Implementation
    }
}

// Start chaincode
shim.start(new MyChaincode());
```

Key Chaincode APIs

Diplomawallah.in

1. State Management APIs

javascript

```
// Write data
await stub.putState(key, Buffer.from(JSON.stringify(value)));

// Read data
let valueBytes = await stub.getState(key);
if (!valueBytes || valueBytes.length === 0) {
    throw new Error('Asset not found');
}
let value = JSON.parse(valueBytes.toString());

// Delete data
await stub.delState(key);
```

2. Query APIs

javascript

```
// Range query (LevelDB and CouchDB)
let iterator = await stub.getStateByRange(startKey, endKey);
let results = [];
while (true) {
    let res = await iterator.next();
    if (res.value && res.value.value.toString()) {
        let record = {
            key: res.value.key,
            value: JSON.parse(res.value.value.toString())
        };
        results.push(record);
    }
    if (res.done) {
```

Diplomawallah.in

```
        await iterator.close();
        break;
    }
}

// Rich query (CouchDB only)
let queryString = {
    selector: {
        owner: 'Alice',
        value: { $gt: 100 }
    }
};
let iterator = await stub.getQueryResult(JSON.stringify(queryString));
```

3. Transaction Context APIs

javascript

```
// Get transaction ID
let txId = stub.getTxID();

// Get transaction timestamp
let timestamp = stub.getTxTimestamp();

// Get channel ID
let channelId = stub.getChannelID();

// Get creator (submitter) identity
let creator = stub.getCreator();
```

4. Access Control APIs

javascript

```
// Get client identity (requires fabric-shim-crypto)
```

Diplomawallah.in

```
const ClientIdentity = require('fabric-shim').ClientIdentity;
let cid = new ClientIdentity(stub);

// Get submitter's MSP ID
let mspId = cid.getMSPID();

// Get submitter's identity attributes
let attrValue = cid.getAttributeValue('role');

// Check if submitter has specific attribute
let hasAttr = cid.assertAttributeValue('role', 'admin');
```

5. Private Data APIs

javascript

```
// Write to private data collection
await stub.putPrivateData(collection, key, value);

// Read from private data collection
let value = await stub.getPrivateData(collection, key);

// Delete from private data collection
await stub.delPrivateData(collection, key);
```

6. Composite Key APIs

javascript

```
// Create composite key
let compositeKey = stub.createCompositeKey('assetType', [owner, assetId]);
await stub.putState(compositeKey, value);

// Query by partial composite key
let iterator = await stub.getStateByPartialCompositeKey('assetType', [owner]);
```

Diplomawallah.in

7. Event APIs

javascript

```
// Emit chaincode event
stub.setEvent('AssetCreated', Buffer.from(JSON.stringify({
    assetId: 'asset1',
    owner: 'Alice',
    timestamp: new Date().toISOString()
})));
```

Chaincode Implementation Example

Complete Asset Management Chaincode (Node.js):

javascript

```
const shim = require('fabric-shim');

class AssetChaincode {

    async Init(stub) {
        console.info('========= Asset Chaincode Init =========');
        return shim.success();
    }

    async Invoke(stub) {
        let ret = stub.getFunctionAndParameters();
        let method = this[ret.fcn];

        if (!method) {
            return shim.error('Function not found: ' + ret.fcn);
        }

        try {
```

Diplomawallah.in

```
        let response = await method(stub, ret.params);
        return shim.success(response);
    } catch (err) {
        console.error(err);
        return shim.error(err.message);
    }
}

// Create a new asset
async createAsset(stub, args) {
    if (args.length !== 4) {
        throw new Error('Incorrect number of arguments. Expecting 4');
    }

    let assetId = args[0];
    let owner = args[1];
    let value = parseInt(args[2]);
    let description = args[3];

    // Check if asset already exists
    let assetBytes = await stub.getState(assetId);
    if (assetBytes && assetBytes.length > 0) {
        throw new Error('Asset already exists: ' + assetId);
    }

    // Create asset object
    let asset = {
        docType: 'asset',
        assetId: assetId,
```

Diplomawallah.in

```
        owner: owner,
        value: value,
        description: description,
        timestamp: stub.getTxTimestamp().seconds.low
    };

    // Save to ledger
    await stub.putState(assetId, Buffer.from(JSON.stringify(asset)));

    // Emit event
    stub.setEvent('AssetCreated', Buffer.from(JSON.stringify(asset)));

    console.info('Asset created: ' + assetId);
    return Buffer.from(JSON.stringify(asset));
}

// Read an asset
async readAsset(stub, args) {
    if (args.length !== 1) {
        throw new Error('Incorrect number of arguments. Expecting assetId');
    }

    let assetId = args[0];
    let assetBytes = await stub.getState(assetId);

    if (!assetBytes || assetBytes.length === 0) {
        throw new Error('Asset not found: ' + assetId);
    }
```

Diplomawallah.in

```
    return assetBytes;
  }

  // Update asset owner
  async transferAsset(stub, args) {
    if (args.length !== 2) {
      throw new Error('Incorrect number of arguments. Expecting assetId and newOwner');
    }

    let assetId = args[0];
    let newOwner = args[1];

    // Get existing asset
    let assetBytes = await stub.getState(assetId);
    if (!assetBytes || assetBytes.length === 0) {
      throw new Error('Asset not found: ' + assetId);
    }

    let asset = JSON.parse(assetBytes.toString());

    // Update owner
    asset.owner = newOwner;
    asset.timestamp = stub.getTxTimestamp().seconds.low;

    // Save updated asset
    await stub.putState(assetId, Buffer.from(JSON.stringify(asset)));

    // Emit event
```

Diplomawallah.in

```
    stub.setEvent('AssetTransferred', Buffer.from(JSON.stringify(asset)));

    console.info('Asset transferred: ' + assetId + ' to ' + newOwner);
    return Buffer.from(JSON.stringify(asset));
}

// Query assets by owner
async queryAssetsByOwner(stub, args) {
    if (args.length !== 1) {
        throw new Error('Incorrect number of arguments. Expecting owner');
    }

    let owner = args[0];

    // Build query (CouchDB)
    let queryString = {
        selector: {
            docType: 'asset',
            owner: owner
        }
    };

    let iterator = await stub.getQueryResult(JSON.stringify(queryString));
    let results = [];

    while (true) {
        let res = await iterator.next();

        if (res.value && res.value.value.toString()) {
```

Diplomawallah.in

```
            let asset = JSON.parse(res.value.value.toString());
            results.push(asset);
        }

        if (res.done) {
            await iterator.close();
            break;
        }
    }

    return Buffer.from(JSON.stringify(results));
}

// Get asset history
async getAssetHistory(stub, args) {
    if (args.length !== 1) {
        throw new Error('Incorrect number of arguments. Expecting assetId');
    }

    let assetId = args[0];
    let iterator = await stub.getHistoryForKey(assetId);
    let history = [];

    while (true) {
        let res = await iterator.next();

        if (res.value) {
            let record = {
                txId: res.value.tx_id,
```

Diplomawallah.in

```
                timestamp: res.value.timestamp,
                isDelete: res.value.is_delete,
                value: res.value.value.toString()
            };
            history.push(record);
        }

        if (res.done) {
            await iterator.close();
            break;
        }
    }

    return Buffer.from(JSON.stringify(history));
  }
}

shim.start(new AssetChaincode());
```

Chaincode Deployment Process

Step 1: Package Chaincode

bash

```
# Navigate to chaincode directory
cd chaincode/asset-chaincode

# Install dependencies
npm install

# Package chaincode
peer lifecycle chaincode package asset.tar.gz \
```

Diplomawallah.in

```
  --path chaincode/asset-chaincode \
  --lang node \
  --label asset_1.0
```

Step 2: Install on Peers

bash

```
# Install on Org1 peer
peer lifecycle chaincode install asset.tar.gz

# Install on Org2 peer (repeat for all orgs)
peer lifecycle chaincode install asset.tar.gz
```

Step 3: Approve Chaincode Definition

bash

```
# Each organization approves
peer lifecycle chaincode approveformyorg \
  --channelID mychannel \
  --name asset \
  --version 1.0 \
  --package-id asset_1.0:hash \
  --sequence 1 \
  --tls --cafile orderer-ca.pem
```

Step 4: Commit Chaincode

bash

```
# After sufficient approvals, commit to channel
peer lifecycle chaincode commit \
  --channelID mychannel \
  --name asset \
  --version 1.0 \
  --sequence 1 \
  --tls --cafile orderer-ca.pem \
```

Diplomawallah.in

```
  --peerAddresses peer0.org1:7051 --tlsRootCertFiles org1-ca.pem \
  --peerAddresses peer0.org2:7051 --tlsRootCertFiles org2-ca.pem
```

Step 5: Invoke Chaincode

```
bash
# Create asset
peer chaincode invoke \
  -o orderer:7050 \
  -C mychannel \
  -n asset \
  -c '{"function":"createAsset","Args":["asset1","Alice","1000","Laptop"]}' \
  --tls --cafile orderer-ca.pem

# Query asset
peer chaincode query \
  -C mychannel \
  -n asset \
  -c '{"function":"readAsset","Args":["asset1"]}'
```

Diplomawallah.in

Chaincode Best Practices

1. Input Validation

- Always validate argument count and types
- Check for empty or null values
- Sanitize inputs to prevent injectionaws.amazon

2. Error Handling

- Use try-catch blocks
- Return descriptive error messages
- Log errors for debugginghyperledger.github+1

3. Access Control

- Verify caller identity before sensitive operations
- Use attribute-based access control (ABAC)

- Implement role-based permissionsaws.amazon

4. Performance

- Minimize state reads/writes
- Use composite keys for efficient queries
- Leverage private data collections for sensitive informationaws.amazon

5. Testing

- Unit test individual functions
- Integration test on test network
- Simulate various scenarios and edge caseshyperledger-fabric.readthedocs+1

6. Documentation

- Comment complex logic
- Document function parameters and return values
- Maintain API documentationhyperledger-fabric.readthedocs

Hyperledger Fabric II: Beyond Chaincode - Fabric SDK and Front-End

While chaincode defines the business logic on the blockchain, applications need interfaces to interact with the network. Hyperledger Fabric provides Software Development Kits (SDKs) that enable developers to build client applications that submit transactions, query ledgers, and listen for events.hyperledger.github+3

Diplomawallah.in

Hyperledger Fabric SDKs

Available SDKs:

1. **Node.js SDK** (Most mature and widely used)hyperledger.github+3
2. **Java SDK**hyperledger-fabric.readthedocs+1
3. **Go SDK**hyperledger-fabric.readthedocs+1
4. **Python SDK** (Under development)oak-tree+1

Note: As of Fabric v2.5, traditional SDKs are deprecated in favor of the **Fabric Gateway client API**, which provides a simpler programming model.github+2

Node.js SDK (fabric-network)

Installation:

bash

npm install fabric-network

Key packages:

- fabric-network: High-level API for application development
- fabric-common: Low-level API for fine-grained interactions
- fabric-ca-client: Client for Fabric Certificate Authority
- fabric-protos: Protocol buffers for gRPC communicationhyperledger.github+1

SDK Components and APIs

1. Connection Profile

Definition: JSON or YAML file describing network topology, including peers, orderers, CAs, and channels.stackoverflow+2

Example (connection-profile.json):

json

```
{
  "name": "test-network",
  "version": "1.0.0",
  "client": {
    "organization": "Org1",
    "connection": {
      "timeout": {
        "peer": { "endorser": "300" },
        "orderer": "300"
      }
    }
  },
  "organizations": {
    "Org1": {
      "mspid": "Org1MSP",
      "peers": ["peer0.org1.example.com"],
      "certificateAuthorities": ["ca.org1.example.com"]
```

Diplomawallah.in

```
        }
    },
    "peers": {
        "peer0.org1.example.com": {
            "url": "grpcs://localhost:7051",
            "tlsCACerts": {
                "path": "crypto/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example.com-cert.pem"
            }
        }
    },
    "orderers": {
        "orderer.example.com": {
            "url": "grpcs://localhost:7050",
            "tlsCACerts": {
                "path": "crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem"
            }
        }
    },
    "certificateAuthorities": {
        "ca.org1.example.com": {
            "url": "https://localhost:7054",
            "caName": "ca-org1"
        }
    }
}
```

Diplomawallah.in

2. Wallet

Purpose: Securely store user identities (certificates and private keys).stackoverflow+2

Types:

- File system wallet
- In-memory wallet (for testing)
- CouchDB wallet
- Redis wallet (for distributed applications)hyperledger.github

Example:

javascript

```
const { Wallets } = require('fabric-network');

// Create file system wallet
const walletPath = path.join(__dirname, 'wallet');
const wallet = await Wallets.newFileSystemWallet(walletPath);

// Add identity to wallet
const identity = {
    credentials: {
        certificate: certPEM,
        privateKey: keyPEM
    },
    mspId: 'Org1MSP',
    type: 'X.509'
};
await wallet.put('user1', identity);

// Retrieve identity from wallet
const userIdentity = await wallet.get('user1');
```

3. Gateway

Diplomawallah.in

Purpose: Main entry point for connecting to the Fabric network and accessing channels.hyperledger-fabric.readthedocs+2

Workflow:

1. Load connection profile
2. Connect to gateway using identity from wallet
3. Access channel
4. Get contract (chaincode)
5. Submit transactions or queries
6. Disconnectdev+2

Example:

javascript

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

async function main() {
  try {
    // Load connection profile
    const ccpPath = path.resolve(__dirname, 'connection-profile.json');
    const ccp = JSON.parse(fs.readFileSync(ccpPath, 'utf8'));

    // Create wallet
    const walletPath = path.join(__dirname, 'wallet');
    const wallet = await Wallets.newFileSystemWallet(walletPath);

    // Check if identity exists in wallet
    const identity = await wallet.get('user1');
    if (!identity) {
      console.log('Identity not found in wallet');
```

Diplomawallah.in

```
    return;
}

// Create gateway instance
const gateway = new Gateway();

// Connect to gateway
await gateway.connect(ccp, {
    wallet: wallet,
    identity: 'user1',
    discovery: { enabled: true, asLocalhost: true }
});

// Get network (channel)
const network = await gateway.getNetwork('mychannel');

// Get contract (chaincode)
const contract = network.getContract('asset');

// Submit transaction
console.log('Submit transaction: createAsset');
await contract.submitTransaction('createAsset', 'asset1', 'Alice', '1000', 'Laptop');
console.log('Transaction submitted successfully');

// Query ledger
console.log('Query: readAsset');
const result = await contract.evaluateTransaction('readAsset', 'asset1');
console.log('Query result:', result.toString());
```

Diplomawallah.in

```
    // Disconnect
    await gateway.disconnect();

  } catch (error) {
    console.error('Error:', error);
    process.exit(1);
  }
}

main();
```

4. Contract (Chaincode Interface)

Purpose: Represents a deployed chaincode, providing methods to submit transactions and queries.hyperledger-fabric.readthedocs+2

Key methods:

javascript

```
// Submit transaction (updates ledger)
await contract.submitTransaction('functionName', 'arg1', 'arg2', ...);

// Evaluate transaction (query, no ledger update)
const result = await contract.evaluateTransaction('functionName', 'arg1', 'arg2', ...);

// Create transaction object for finer control
const transaction = contract.createTransaction('functionName');
transaction.setEndorsingOrganizations('Org1MSP', 'Org2MSP');
await transaction.submit('arg1', 'arg2');
```

5. Event Listeners

Purpose: Listen for chaincode events, block events, or transaction events.dev+1

Example:

javascript

Diplomawallah.in

```
// Listen for chaincode events
const listener = async (event) => {
    console.log('Event received:', event.eventName);
    console.log('Payload:', event.payload.toString());
};

await contract.addContractListener(listener);

// Listen for block events
const blockListener = async (event) => {
    console.log('Block number:', event.blockNumber);
    console.log('Transactions:', event.blockData.data.data.length);
};

await network.addBlockListener(blockListener);

// Remove listeners
contract.removeContractListener(listener);
```

Diplomawallah.in

Building a Front-End Application

Front-end applications interact with Hyperledger Fabric through a **backend REST API** built with the Fabric SDK.stackoverflow+2

Architecture:

text

[Front-End (React/Angular/Vue)]

↓ HTTP/REST

[Backend API Server (Express.js + Fabric SDK)]

↓ gRPC

[Hyperledger Fabric Network]

Backend REST API Example (Node.js + Express)

Project structure:

text

```
application/
├── server.js              # Express server
├── fabric/
│   ├── network.js         # Fabric network interaction
│   ├── wallet/            # User identities
│   └── connection-profile.json
├── routes/
│   └── asset.js           # API routes
└── package.json
```

server.js:

javascript

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const assetRoutes = require('./routes/asset');

const app = express();
const PORT = 3000;

// Middleware
app.use(cors());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// Routes
app.use('/api/assets', assetRoutes);
```

Diplomawallah.in

```
// Health check
app.get('/health', (req, res) => {
  res.json({ status: 'OK' });
});

// Start server
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

fabric/network.js:

javascript

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

class FabricNetwork {
  constructor() {
    this.ccpPath = path.resolve(__dirname, 'connection-profile.json');
    this.walletPath = path.join(__dirname, 'wallet');
  }

  async connectToNetwork(userId) {
    try {
      // Load connection profile
      const ccp = JSON.parse(fs.readFileSync(this.ccpPath, 'utf8'));

      // Load wallet
      const wallet = await Wallets.newFileSystemWallet(this.walletPath);
```

Diplomawallah.in

```
        // Check identity
        const identity = await wallet.get(userId);
        if (!identity) {
            throw new Error(`Identity ${userId} not found in wallet`);
        }

        // Connect to gateway
        const gateway = new Gateway();
        await gateway.connect(ccp, {
            wallet: wallet,
            identity: userId,
            discovery: { enabled: true, asLocalhost: true }
        });

        // Get network and contract
        const network = await gateway.getNetwork('mychannel');
        const contract = network.getContract('asset');

        return { gateway, network, contract };

    } catch (error) {
        console.error('Failed to connect to network:', error);
        throw error;
    }
}

async disconnectFromNetwork(gateway) {
    if (gateway) {
        await gateway.disconnect();
```

Diplomawallah.in

```
    }
  }
}

module.exports = FabricNetwork;
```

routes/asset.js:

javascript

```
const express = require('express');
const router = express.Router();
const FabricNetwork = require('../fabric/network');

const fabricNetwork = new FabricNetwork();

// Create asset
router.post('/', async (req, res) => {
  try {
    const { assetId, owner, value, description } = req.body;
    const userId = req.headers['user-id'] || 'user1';

    const { gateway, contract } = await fabricNetwork.connectToNetwork(userId);

    await contract.submitTransaction('createAsset', assetId, owner, value,
description);

    await fabricNetwork.disconnectFromNetwork(gateway);

    res.json({
      success: true,
      message: 'Asset created successfully',
```

Diplomawallah.in

```
            assetId: assetId
        });

    } catch (error) {
        console.error('Error creating asset:', error);
        res.status(500).json({
            success: false,
            message: error.message
        });
    }
});

// Get asset
router.get('/:assetId', async (req, res) => {
    try {
        const { assetId } = req.params;
        const userId = req.headers['user-id'] || 'user1';

        const { gateway, contract } = await fabricNetwork.connectToNetwork(userId);

        const result = await contract.evaluateTransaction('readAsset', assetId);

        await fabricNetwork.disconnectFromNetwork(gateway);

        res.json({
            success: true,
            data: JSON.parse(result.toString())
        });
```

Diplomawallah.in

```
    } catch (error) {
        console.error('Error reading asset:', error);
        res.status(500).json({
            success: false,
            message: error.message
        });
    }
});

// Transfer asset
router.put('/:assetId/transfer', async (req, res) => {
    try {
        const { assetId } = req.params;
        const { newOwner } = req.body;
        const userId = req.headers['user-id'] || 'user1';

        const { gateway, contract } = await fabricNetwork.connectToNetwork(userId);

        await contract.submitTransaction('transferAsset', assetId, newOwner);

        await fabricNetwork.disconnectFromNetwork(gateway);

        res.json({
            success: true,
            message: 'Asset transferred successfully'
        });

    } catch (error) {
        console.error('Error transferring asset:', error);
```

DiplomaWallah.in

```
        res.status(500).json({
            success: false,
            message: error.message
        });
    }
});

// Query assets by owner
router.get('/owner/:owner', async (req, res) => {
    try {
        const { owner } = req.params;
        const userId = req.headers['user-id'] || 'user1';

        const { gateway, contract } = await fabricNetwork.connectToNetwork(userId);

        const result = await contract.evaluateTransaction('queryAssetsByOwner',
owner);

        await fabricNetwork.disconnectFromNetwork(gateway);

        res.json({
            success: true,
            data: JSON.parse(result.toString())
        });

    } catch (error) {
        console.error('Error querying assets:', error);
        res.status(500).json({
            success: false,
```

Diplomawallah.in

```
            message: error.message
        });
    }
});

module.exports = router;
```

Front-End Example (React)

Component to interact with REST API:

javascript

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const API_BASE = 'http://localhost:3000/api';

function AssetManager() {
    const [assets, setAssets] = useState([]);
    const [assetId, setAssetId] = useState('');
    const [owner, setOwner] = useState('');
    const [value, setValue] = useState('');
    const [description, setDescription] = useState('');

    // Fetch assets
    const fetchAssets = async (ownerName) => {
        try {
            const response = await axios.get(
                `${API_BASE}/assets/owner/${ownerName}`,
                { headers: { 'user-id': 'user1' } }
            );
            setAssets(response.data.data);
```

Diplomawallah.in

```
    } catch (error) {
      console.error('Error fetching assets:', error);
    }
  };

  // Create asset
  const createAsset = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post(
        `${API_BASE}/assets`,
        { assetId, owner, value, description },
        { headers: { 'user-id': 'user1' } }
      );
      alert('Asset created: ' + response.data.assetId);
      // Clear form
      setAssetId('');
      setOwner('');
      setValue('');
      setDescription('');
    } catch (error) {
      console.error('Error creating asset:', error);
      alert('Error: ' + error.response.data.message);
    }
  };

  return (
    <div>
      <h1>Asset Management</h1>
```

Diplomawallah.in

```
        <h2>Create Asset</h2>
        <form onSubmit={createAsset}>
          <input placeholder="Asset ID" value={assetId} onChange={(e) =>
setAssetId(e.target.value)} required />
          <input placeholder="Owner" value={owner} onChange={(e) =>
setOwner(e.target.value)} required />
          <input placeholder="Value" value={value} onChange={(e) =>
setValue(e.target.value)} required />
          <input placeholder="Description" value={description} onChange={(e) =>
setDescription(e.target.value)} required />
          <button type="submit">Create</button>
        </form>

        <h2>Query Assets</h2>
        <button onClick={() => fetchAssets('Alice')}>Show Alice's Assets</button>

        <ul>
          {assets.map((asset, idx) => (
            <li key={idx}>
              {asset.assetId}: {asset.description} (Owner: {asset.owner}, Value:
{asset.value})
            </li>
          ))}
        </ul>
      </div>
  );
}

export default AssetManager;
```

SDK Best Practices

Diplomawallah.in

1. Connection Management

- Reuse gateway connections when possible
- Close gateway connections after use
- Implement connection pooling for high-traffic applicationshyperledger-fabric.readthedocs+1

2. Error Handling

- Wrap SDK calls in try-catch blocks
- Handle network errors gracefully
- Provide meaningful error messages to usershyperledger.github+1

3. Identity Management

- Store credentials securely
- Implement proper authentication
- Use role-based access controlhyperledger.github

4. Performance

- Use evaluateTransaction for queries (doesn't invoke consensus)
- Use submitTransaction only for ledger updates
- Batch transactions when possiblehyperledger-fabric.readthedocs+1

5. Event Handling

- Always remove event listeners when done
- Handle listener errors appropriately
- Consider event replay for missed eventshyperledger.github

Diplomawallah.in

Hyperledger Composer Tool

Note: As of August 29, 2019, **Hyperledger Composer has been officially deprecated** and is no longer actively maintained.stackoverflow+4

Deprecation reasons:

1. **Dual programming models:** Composer created confusion by offering a separate programming model from native Fabric chaincode.stackoverflow+1
2. **Feature lag:** Difficult to keep pace with latest Fabric features (e.g., private data collections).stackoverflow

3. **Limited adoption:** Most users reverted to native Fabric development after POC stage.stackoverflow+1

Recommendation: Use **Hyperledger Fabric v1.4+** native development instead, which offers significant improvements to the developer experience.github+3

What Was Hyperledger Composer?

Definition: Hyperledger Composer was an application development framework designed to simplify and accelerate the creation of Hyperledger Fabric blockchain applications.github+3

Target users: Business analysts and developers without deep blockchain expertise.geeksforgeeks+1

Key Concepts in Hyperledger Composer

1. Business Network Archive (BNA)

- Central artifact containing all business network components
- Packaged as .bna file
- Included model files, transaction logic, access control rulesinvestopedia+1

2. Assets

- Resources that can be exchanged (e.g., properties, vehicles)
- Defined with unique identifiers and attributes
- Could be tangible or intangiblegeeksforgeeks

3. Participants

- Members of the business network (e.g., buyers, sellers)
- Had unique identities
- Could own assets and participate in transactionsgeeksforgeeks

4. Transactions

- Operations that change asset states
- Defined business logic for asset exchanges
- Automatically recorded on blockchaingeeksforgeeks

5. Access Control

- Rules defining who can do what
- Specified in Access Control Language (ACL)
- Enforced by Composer runtimegeeksforgeeks

Diplomawallah.in

6. Queries

- Predefined or ad-hoc data retrieval
- Expressed in Composer Query Language
- Returned filtered resultsgeeksforgeeks

7. Events

- Notifications emitted by transactions
- Could trigger external system actions
- Subscribed to by client applicationsgeeksforgeeks

Composer Architecture Components

1. Modeling Language

- Domain-specific language for defining business networks
- .cto files (CTO = Composer Type Object)
- Declarative syntaxinvestopedia+1

Example:

text

```
namespace org.example

asset Car identified by vin {
  o String vin
  o String make
  o String model
  --> Person owner
}

participant Person identified by email {
  o String email
  o String firstName
  o String lastName
}
```

Diplomawallah.in

transaction TransferCar {

--> Car car

--> Person newOwner

}

2. Transaction Processor Functions

- JavaScript functions implementing transaction logic
- Automatically invoked when transactions submittedgeeksforgeeks

3. Access Control Language

- Declarative rules in .acl files
- Specified permissions for participantsgeeksforgeeks

4. Composer Playground

- Web-based tool for developing and testing business networks
- No installation required
- Rapid prototyping environmentinvestopedia+1

5. Composer REST Server

- Automatically generated REST API for business network
- Swagger-documented endpoints
- Enabled rapid front-end developmentinvestopedia+1

6. Composer CLI

- Command-line tools for managing business networks
- Deploy, update, test networksgeeksforgeeks

Why Composer Was Deprecated

Despite initial popularity for rapid prototyping, Composer faced fundamental limitations:stackoverflow+1

1. Abstraction Cost

- Abstracted away Fabric concepts, creating learning gap
- Users struggled to transition from Composer to production-grade Fabricstackoverflow+1

2. Maintenance Burden

Diplomawallah.in

- Required constant updates to match Fabric evolution
- Difficult to expose new Fabric features through Composer abstractionstackoverflow

3. Production Concerns

- Not suitable for production deployments
- Lacked full feature set of native Fabricstackoverflow+1
- Performance overhead from abstraction layerstackoverflow

4. Governance Model Mismatch

- Composer's centralized governance model didn't align with Fabric 2.x's decentralized chaincode lifecyclestackoverflow

Migration Path

Organizations using Composer should migrate to native Hyperledger Fabric development:github+2

Recommended approach:

1. **Understand native Fabric:** Learn chaincode development in Go, Node.js, or Javahyperledger-fabric.readthedocs+2
2. **Rewrite business logic:** Convert Composer transaction processors to chaincode functionshyperledger-fabric.readthedocs
3. **Implement access control:** Use Fabric's built-in MSP and attribute-based access controlhyperledger-fabric.readthedocs+1
4. **Build client applications:** Use Fabric SDKs directlygithub+2
5. **Test thoroughly:** Validate migrated functionality on test networkhyperledger-fabric.readthedocs+1

Resources for migration:

- Official Fabric documentation
- Fabric samples repository
- Commercial paper tutorial (good starting point)hyperledger-fabric.readthedocs+1

Summary of Key Concepts

This unit covered the architecture and development aspects of Hyperledger Fabric:

Diplomawallah.in

1. Decomposing the Consensus Process: Examined how consensus mechanisms can be broken into distinct phases (transaction proposal, broadcast, validation, voting, finalization, propagation) and components (election, proposal/acceptance, agreement, commitment). Explored Fabric's unique execute-order-validate model.

2. Hyperledger Fabric Components: Detailed study of seven core components:

- **Peers:** Host ledgers and chaincode; endorsing, committing, anchor, and leader types
- **Orderers:** Order transactions and create blocks using Raft consensus
- **Chaincode:** Smart contracts written in Go, Node.js, or Java
- **MSP:** Manages identities and access control via X.509 certificates
- **Channels:** Enable private transactions with separate ledgers
- **Ledger:** Consists of blockchain (transaction log) and world state (current values)
- **Certificate Authority:** Issues and manages digital certificates

3. Chaincode Design and Implementation: Covered design principles (determinism, statelessness, modularity, error handling, security), key APIs (state management, queries, transaction context, access control, private data, events), complete implementation examples, and deployment process.

4. Fabric SDK and Front-End Development: Explored Node.js SDK (fabric-network package), key components (connection profile, wallet, gateway, contract), event listeners, building backend REST APIs with Express.js, and integrating front-end applications (React/Angular/Vue) with blockchain backend.

5. Hyperledger Composer Tool: Historical perspective on the deprecated Composer framework, its key concepts (assets, participants, transactions, BNA), architecture components, reasons for deprecation, and migration path to native Fabric development.

Diplomawallah.in

Practice Questions

Question 1: Explain Hyperledger Fabric's execute-order-validate consensus model and compare it to the traditional order-execute model used by Bitcoin and Ethereum. Discuss the advantages of Fabric's approach for enterprise blockchain applications.

Question 2: Describe the complete transaction flow in Hyperledger Fabric from client proposal to ledger update. Include the roles of endorsing peers, ordering service, and committing peers, and explain how the Membership Service Provider (MSP) ensures security throughout the process.

Diploma Wallah

Made with 💗 by Sangam

Diplomawallah.in