Over All network Structure

Based on your request, you want a step-by-step guide to set up a Hyperledger Fabric network using Node.js/JavaScriptfor all configuration and interaction steps, without using Go or Java, and without directly providing bash scripts. This implies using the Fabric Node.js SDK for network configuration tasks (like creating channels, joining peers, installing chaincode) instead of the standard Fabric CLI commands or the network.sh script.
This is a more advanced approach as the standard test-network scripts simplify many underlying CLI commands. However, it's definitely achievable with the Node.js SDK.
Here's a detailed, step-by-step guide focusing on what actions you need to perform using Node.js/JavaScript, without direct shell scripts.

Part 1: Prerequisites and Environment Setup

What to do:
Install Essential Tools:
Docker Desktop (or Docker Engine + Docker Compose): Download and install from the official Docker website. Ensure Docker is running.
Node.js (LTS version, e.g., 18.x or 20.x): Install via your system's package manager or from the official Node.js website.
npm (Node Package Manager): This usually comes with Node.js.
Git: Install Git.
cURL: Install cURL.
Download Fabric Binaries and Samples:
Open your terminal.
Create a directory for your Fabric development (e.g., fabric-dev).
Navigate into this directory.
Use curl to download the install-fabric.sh script from the official Hyperledger Fabric GitHub.
Execute this script with specific Fabric and Fabric CA versions (e.g., 2.5.0 for Fabric, 1.5.0 for Fabric CA) and specify docker to pull Docker images.
Manually add the bin directory from the downloaded fabric-samples to your system's PATH environment variable so you can execute Fabric binaries like cryptogen and configtxgen directly. This usually involves editing your shell's profile file (e.g., .bashrc, .zshrc) and then sourcing it or restarting your terminal.

Part 2: Generating Cryptographic Material and Genesis Block (using Fabric Binaries)

While the goal is to use Node.js, core tasks like generating crypto material and the genesis block are typically done once using the Fabric binaries (cryptogen, configtxgen) as they are fundamental setup steps before the network even starts. The Node.js SDK primarily interacts with a running network.
What to do:
Create Project Directory Structure:
Inside your fabric-dev directory, create a new project folder (e.g., my-fabric-network-js).
Inside my-fabric-network-js, create subdirectories like organizations, channel-artifacts.
Define Network Topology (Configuration Files):
crypto-config.yaml: Create this file in your project's root. Define your organizations (e.g., OrdererOrg, Org1, Org2), their CAs, and the number of peers for each organization. This file describes the identities needed.
configtx.yaml: Create this file. Define your orderer type (e.g., Raft), the consortiums (groups of organizations that can create channels), and channel profiles (e.g., TwoOrgChannel profile for your application channel). This file describes the network structure.
Generate Cryptographic Material:
Open your terminal, navigate to my-fabric-network-js.
Execute the cryptogen tool using your crypto-config.yaml file to generate certificates and private keys for all your organizations, CAs, orderers, and peers. These will be placed in the organizations directory.

Generate Orderer Genesis Block:
Using the configtxgen tool, create the genesis block for your ordering service. This block is the first block in the orderer's ledger and contains initial network configuration. Save it in your channel-artifactsdirectory.
Generate Channel Transaction Configuration:
Using the configtxgen tool again, create the channel configuration transaction file (e.g., mychannel.tx). This file contains the initial configuration for your application channel. Save it in your channel-artifactsdirectory.

Part 3: Defining and Starting Network Components (using Docker Compose)

What to do:
Create Docker Compose File:
In your project's root (my-fabric-network-js), create a docker-compose.yaml file.
Define Docker services for:
Orderer(s): One or more orderer nodes (e.g., orderer.example.com).
Certificate Authorities (CAs): One CA for each organization (e.g., ca_org1, ca_org2).
Peers: One or more peer nodes for each organization (e.g., peer0.org1.example.com, peer0.org2.example.com).
(Optional) CouchDB/LevelDB: If using CouchDB as a state database, include a service for each peer that will use it.
Configure Docker Compose Services:
For each service, specify:
The Docker image (e.g., hyperledger/fabric-peer:2.5.0).
Environment variables (e.g., CORE_PEER_MSPCONFIGPATH, CORE_PEER_LOCALMSPID, paths to crypto material, database settings).
Volumes to mount your crypto material and chaincode.
Ports to expose.
Dependencies between services.
Start Docker Containers:
Open your terminal, navigate to my-fabric-network-js.
Start all the services defined in your docker-compose.yaml file in detached mode. This will bring up your orderers, CAs, and peers.
Verify that all containers are running.

Part 4: Network Configuration and Chaincode Deployment (using Node.js SDK)

This is where the Node.js/JavaScript interaction truly begins. You'll create Node.js scripts for each step.
Directory Structure for Node.js Scripts:

```
my-fabric-network-js/
 ├─── organizations/ (crypto material generated by cryptogen)
 ├─── channel-artifacts/ (genesis block, channel tx generated by configtxgen)
 ├─── docker-compose.yaml
 └─── scripts/
     ├─── helpers.js (utility functions like getCCP, getWallet, etc.)
     ├─── enrollAdmin.js
     ├─── registerUser.js
     ├─── createChannel.js
     ├─── joinPeersToChannel.js
     ├─── updateChannelAnchorPeers.js
     ├─── packageChaincode.js
     ├─── installChaincode.js
```

```
    ├─── approveChaincode.js
    ├─── commitChaincode.js
    ├─── invokeChaincode.js
    └─── queryChaincode.js
```
What to do (Node.js/JavaScript Scripts):
Install Fabric Node.js SDK:
In your my-fabric-network-js directory, initialize a Node.js project.
Install the fabric-network and fabric-ca-client packages.
Create Connection Profiles (CCP files):
Manually create JSON files (e.g., connection-org1.json, connection-org2.json) that describe how client applications can connect to your Fabric network. These files should contain details like peer endpoints, orderer endpoints, CA endpoints, and TLS certificates. The fabric-samples test-networkprovides examples of these that you can adapt. Place these in your organizations directory alongside your crypto material.
Enroll Admin for Each Organization (Script: enrollAdmin.js):
For each organization (Org1, Org2), write a Node.js script.
Purpose: To enroll the administrator user (e.g., admin) with that organization's Certificate Authority (CA). This admin identity will be used to register and enroll other users, and perform administrative tasks.
Steps:
Load the connection profile (CCP) for the organization.
Create a new File System Wallet.
Create a Fabric CA Client instance.
Use the CA client to enroll the admin user (e.g., admin, with admin secret).
Store the enrolled admin's identity in the wallet.
Register and Enroll Application User (Script: registerUser.js):
Write a Node.js script.
Purpose: To register and enroll an application user (e.g., appUser) under an organization's admin. This appUser identity will be used by your client applications to interact with chaincode.
Steps:
Load the CCP for the organization.
Load the admin identity from the wallet.
Create a new Gateway instance and connect to the network using the admin identity.
Get the CA service from the connected network.
Register the new user with the CA using the admin's identity.
Enroll the registered user with the CA.
Store the enrolled user's identity in the wallet.
Create Channel (Script: createChannel.js):
Write a Node.js script.
Purpose: To create your application channel (e.g., mychannel).
Steps:
Load the CCP for one of the organizations that will be part of the channel (e.g., Org1).
Load the admin identity for that organization.
Create a Gateway instance and connect to the network using the admin identity.
Obtain a network object representing the "system channel" (the channel where the ordering service configuration resides, though in current Fabric, it's often implied rather than explicitly interacted with for channel creation).
Use the configtxgen tool (from your PATH) to generate the mychannel.tx (channel creation transaction) file before running this script.
Use the Channel methods from the SDK (specifically, interacting with the orderer directly or via the gateway's client object) to send the channel creation transaction, pointing to your mychannel.txfile.
Join Peers to Channel (Script: joinPeersToChannel.js):
Write a Node.js script.
Purpose: To join peers from participating organizations to the newly created channel.
Steps:
For each organization whose peer needs to join:
Load its CCP and admin identity.
Create a Gateway instance and connect.

Get a reference to the mychannel object.
Use the channel.joinPeer() method, specifying the peer to join (e.g., peer0.org1.example.com).
You might need to retrieve the peer object from the client configuration.
Repeat for all relevant peers (e.g., peer0.org2.example.com).
Update Anchor Peers (Script: updateChannelAnchorPeers.js):
Write a Node.js script.
Purpose: To inform other organizations on the channel about the anchor peers of each organization. Anchor peers enable cross-organization gossip communication.
Steps:
Using configtxgen, generate an anchor peer update transaction for each organization (e.g., Org1MSPanchors.tx, Org2MSPanchors.tx). This will involve a specific profile in your configtx.yaml.
For each organization:
Load its CCP and admin identity.
Create a Gateway instance and connect.
Get a reference to the mychannel object.
Use channel.sendUpdateChannel() method, providing the anchor peer update transaction file.
Package Chaincode (Script: packageChaincode.js):
Write your chaincode in Node.js (e.g., my-chaincode.js and package.json for dependencies).
Place it in a dedicated directory (e.g., chaincode/my-chaincode).
Write a Node.js script.
Purpose: To package your chaincode into a .tgz file, which is the format required for installation on peers.
Steps:
Use the Fabric Node.js SDK's utility functions (e.g., Package.fromDirectory) to package your chaincode source directory.
Save the resulting package to a file (e.g., my-chaincode.tgz).
Install Chaincode on Peers (Script: installChaincode.js):
Write a Node.js script.
Purpose: To install the chaincode package on the peers of each organization that will endorse transactions.
Steps:
For each organization's peer where the chaincode needs to be installed:
Load its CCP and admin identity.
Create a Gateway instance and connect.
Get the client object from the gateway.
Use client.installChaincode() method, passing the chaincode package buffer and a list of target peers.
Capture the packageId returned, as you'll need it for approval and commit steps.
Approve Chaincode Definition (Script: approveChaincode.js):
Write a Node.js script.
Purpose: Each organization needs to approve the chaincode definition (name, version, endorsement policy, packageId) for the channel.
Steps:
For each organization that needs to approve:
Load its CCP and admin identity.
Create a Gateway instance and connect to the mychannel.
Get the network.getContract('_lifecycle') for the system chaincode.
Invoke the _lifecycle system chaincode's ApproveChaincodeDefinitionForMyOrg function, passing the chaincode name, version, sequence number, endorsement policy, and the packageId obtained during installation.
Commit Chaincode Definition (Script: commitChaincode.js):
Write a Node.js script.
Purpose: Once enough organizations (based on the channel's _lifecycle endorsement policy, typically a majority) have approved, one organization can commit the chaincode definition to the channel.
Steps:
Load the CCP and admin identity for one of the approving organizations.
Create a Gateway instance and connect to the mychannel.
Get the network.getContract('_lifecycle').

Invoke the _lifecycle system chaincode's CommitChaincodeDefinition function, passing the chaincode name, version, sequence number, and endorsement policy.
Invoke Chaincode (Script: invokeChaincode.js):
Write a Node.js script.
Purpose: To send transaction proposals to the network that will invoke your custom chaincode functions (e.g., createAsset, transferAsset).
Steps:
Load the CCP for the organization of the user who will invoke (e.g., Org1).
Load the appUser identity.
Create a Gateway instance and connect to mychannel.
Get a reference to your custom chaincode contract (e.g., contract = network.getContract('my-chaincode')).
Use contract.submitTransaction('functionName', 'arg1', 'arg2', ...) to invoke a chaincode function.
Handle transaction response and errors.
Query Chaincode (Script: queryChaincode.js):
Write a Node.js script.
Purpose: To read data from the ledger by invoking a chaincode query function (e.g., readAsset, getAllAssets).
Steps:
Load the CCP and appUser identity.
Create a Gateway instance and connect to mychannel.
Get a reference to your custom chaincode contract.
Use contract.evaluateTransaction('queryFunctionName', 'arg1') to perform a query.
The result is returned directly to the client; it does not go through the ordering service.

Part 5: Specific Use Case: Information Utility (Using Node.js Application)

What to do:
Define Your "Information Utility" Chaincode:
This is the same as in the previous explanation. Develop your Node.js chaincode (e.g., infoUtilityChaincode.js) with functions like createInformationRecord, updateInformationStatus, queryInformation, getInformationHistory.
Ensure your chaincode handles data modeling, access control (e.g., based on ctx.clientIdentity to check roles/MSPs), and error handling.
Modify and Use the Node.js SDK Scripts:
Use the packageChaincode.js, installChaincode.js, approveChaincode.js, commitChaincode.jsscripts to deploy your infoUtilityChaincode.js.
Make sure to update the chaincode name (-ccn), path (-ccp), and language (-ccl) arguments or variables in your Node.js scripts.
Adapt invokeChaincode.js and queryChaincode.js to call the specific functions of your infoUtilityChaincode (e.g., contract.submitTransaction('createInformationRecord', ...) or contract.evaluateTransaction('queryInformation', ...)).
Build a Front-end Application (Optional but Recommended):
Develop a separate web (e.g., React, Angular, Vue) or desktop application.
This application will serve as the user interface for your Information Utility.
It will communicate with a Node.js backend (your "client application") that contains the invokeChaincode.js and queryChaincode.js logic.
The backend will use the Fabric Node.js SDK to interact with the blockchain network on behalf of the users.
Implement user authentication and authorization at the application layer before calling blockchain functions.
Example Flow for Information Utility using Node.js SDK scripts:
Preparation:
Generate crypto and channel artifacts (cryptogen, configtxgen).
Start Docker containers (docker-compose up -d).
Network Setup (Run these Node.js scripts in order):
node scripts/enrollAdmin.js org1
node scripts/registerUser.js org1 appUser
node scripts/enrollAdmin.js org2

```
node scripts/registerUser.js org2 appUser
node scripts/createChannel.js mychannel
node scripts/joinPeersToChannel.js mychannel org1 peer0
node scripts/joinPeersToChannel.js mychannel org2 peer0
node scripts/updateChannelAnchorPeers.js mychannel org1
node scripts/updateChannelAnchorPeers.js mychannel org2
```

Chaincode Deployment (Run these Node.js scripts in order):

```
node scripts/packageChaincode.js infoUtility infoUtilityChaincode.js
node scripts/installChaincode.js infoUtility org1
node scripts/installChaincode.js infoUtility org2
node scripts/approveChaincode.js infoUtility mychannel org1
node scripts/approveChaincode.js infoUtility mychannel org2
node scripts/commitChaincode.js infoUtility mychannel
```

Application Interaction (Run these Node.js scripts):

```
node scripts/invokeChaincode.js infoUtility createInformationRecord "ID001" "Data1" "Org1" ...
node scripts/queryChaincode.js infoUtility queryInformation "ID001"
node scripts/invokeChaincode.js infoUtility updateInformationStatus "ID001" "Verified"
node scripts/queryChaincode.js infoUtility queryInformation "ID001"
```

This approach gives you granular control over each Fabric operation using Node.js, fulfilling your requirement of not using direct Go or Java. Remember that this is more verbose than using network.sh, but it provides a deeper understanding of the underlying SDK interactions.