Hyperledger Fabric Endorsement Policies

By Coding-Bootcamps.com

Here is the outline of topics covered:

Endorsement Policy Overview

Every chaincode has an endorsement policy which specifies the set of peers on a channel that must execute chaincode and endorse the execution results in order for the transaction to be considered valid. These endorsement policies define the organizations (through their peers) who must "endorse" (i.e., approve of) the execution of a proposal.

As part of the transaction validation steps performed by the peers are:

- Each validating peer checks to make sure that the transaction contains the appropriate **number** of endorsements and that
- Transactions are from the expected sources (both of these are specified in the endorsement policy).
- The endorsements are also checked to make sure they're valid (i.e., that they are valid signatures from valid certificates).

1. Multiple ways to require endorsement

By default, endorsement policies are specified in the chaincode definition, which is agreed to by channel members and then committed to a channel (that is, one endorsement policy covers all of the state associated with a chaincode).

For private data collections, you can also specify an endorsement policy at the private data collection level, which would override the chaincode level endorsement policy for any keys in the private data collection, thereby further restricting which organizations can write to a private data collection.

Finally, there are cases where it may be necessary for a particular public channel state or private data collection state (a particular key-value pair, in other words) to have a different endorsement policy. This **state-based endorsement** allows the chaincode-level or collection-level endorsement policies to be overridden by a different policy for the specified keys.

To illustrate the circumstances in which the various types of endorsement policies might be used, consider a channel on which cars are being exchanged. The "creation" — also known as "issuance" – of a car as an asset that can be traded (putting the key-value pair that represents it into the world state, in other words) would have to satisfy the chaincode-level endorsement policy. To see how to set a chaincode-level endorsement policy, check out the section below.

If the key representing the car requires a specific endorsement policy, it can be defined either when the car is created or afterwards. There are a number of reasons why it might be necessary or preferable to set a state-specific endorsement policy. The car might have historical importance or value that makes it necessary to have the endorsement of a licensed appraiser. Also, the owner of the car (if they're a member of the channel) might also want to ensure that their peer signs off on a transaction. In both cases, **an endorsement policy is required for a particular asset that is different from the default endorsement policies for the other assets associated with that chaincode.**

We'll show you how to define a state-based (represented by key-value pairs) endorsement policy in a subsequent section (see section 5 or Setting key-level endorsement policies). But first, let's see how we set a chaincode-level endorsement policy.

2. Setting chaincode-level endorsement policies

Chaincode-level endorsement policies are agreed to by channel members when they approve a chaincode definition for their organization. A sufficient number of channel members need to approve a chaincode definition to meet the **Channel/Application/LifecycleEndorsement** policy, which by default is set to a majority of channel members, before the definition can be committed to the channel. Once the definition has been committed, the chaincode is ready to use. Any invoke of the chaincode that writes data to the ledger will need to be validated by enough channel members to meet the endorsement policy.

You can specify an endorsement policy for a chaincode using the Fabric SDKs. You can also create an endorsement policy from your Command Line Interface when you approve and commit a chaincode definition with the Fabric peer binaries by using the --signature-policy flag.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --signature-policy
"AND('Org1.member', 'Org2.member')" --name mycc --version 1.0 --package-id
mycc_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1
--tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/exampl
e.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --
waitForEvent
```

The above command approves the chaincode definition of mycc with the policy AND('Org1.member', 'Org2.member') which would require that a member of both Org1 and Org2 sign the transaction. After a sufficient number of channel members approve a chaincode definition for mycc, the definition and endorsement policy can be committed to the channel using the command below:

```
peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID mychannel --
signature-policy "AND('Org1.member', 'Org2.member')" --name mycc --version 1.0 --
sequence 1 --init-required --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/exampl
e.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --
waitForEvent --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.exam
ple.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses
peer0.org2.example.com:9051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.exam
ple.com/peers/peer0.org2.example.com/tls/ca.crt
```

Notice that, if the identity classification is enabled, one can use the PEER role to restrict endorsement to only peers.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --signature-policy
"AND('Org1.peer', 'Org2.peer')" --name mycc --version 1.0 --package-id
mycc_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1
--tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/exampl
e.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --
waitForEvent
```

In addition to the specifying an endorsement policy from the CLI or SDK, a chaincode can also use policies in the channel configuration as endorsement policies. You can use the --channel-config-policy flag to select a channel policy with format used by the channel configuration and by ACLs.

For example:

```
peer lifecycle chaincode approveformyorg --channelID mychannel --channel-config-
policy Channel/Application/Admins --name mycc --version 1.0 --package-id
mycc_1:3a8c52d70c36313cfebbaf09d8616e7a6318ababa01c7cbe40603c373bcfe173 --sequence 1
--tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/exampl
e.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem --
waitForEvent
```

If you do not specify a policy, the chaincode definition will use the Channel/Application/Endorsement policy by default, which requires that a transaction be validated by a majority of channel members. This policy depends on the membership of the channel, so it will be updated automatically when organizations are added or removed from a channel. One advantage of using channel policies is that they can be written to be updated automatically with channel membership.

If you specify an endorsement policy using the --signature-policy flag or the SDK, you will need to update the policy when organizations join or leave the channel. A new organization added to the channel after the chaincode has been defined will be able to query a chaincode (provided the query has appropriate authorization as defined by channel policies and any application level checks enforced by the chaincode) but will not be able to execute or endorse the chaincode. Only organizations listed in the endorsement policy syntax will be able sign transactions.

3. Endorsement policy syntax

As you can see above, policies are expressed in terms of principals (“principals” are identities matched to a role). Principals are described as 'MSP.ROLE', where MSP represents the required MSP ID and ROLE represents one of the four accepted roles: member, admin, client, and peer.

Here are a few examples of valid principals:

- 'Org0.admin': any administrator of the Org0 MSP
- 'Org1.member': any member of the Org1 MSP
- 'Org1.client': any client of the Org1 MSP
- 'Org1.peer': any peer of the Org1 MSP

The syntax of the language is:

EXPR(E[, E...])

Where EXPR is either AND, OR, or OutOf, and E is either a principal (with the syntax described above) or another nested call to EXPR.

For example:

- AND('Org1.member', 'Org2.member', 'Org3.member') requests one signature from each of the three principals.
- OR('Org1.member', 'Org2.member') requests one signature from either one of the two principals.
- OR('Org1.member', AND('Org2.member', 'Org3.member')) requests either one signature from a member of the Org1 MSP or one signature from a member of the Org2 MSP and one signature from a member of the Org3 MSP.
- OutOf(1, 'Org1.member', 'Org2.member'), which resolves to the same thing as OR('Org1.member', 'Org2.member').
- Similarly, OutOf(2, 'Org1.member', 'Org2.member') is equivalent to AND('Org1.member', 'Org2.member'), and OutOf(2, 'Org1.member', 'Org2.member', 'Org3.member') is equivalent to OR(AND('Org1.member', 'Org2.member'), AND('Org1.member', 'Org3.member'), AND(' Org2.member', 'Org3.member')).

4. Setting collection-level endorsement policies

Similar to chaincode-level endorsement policies, when you approve and commit a chaincode definition, you can also specify the chaincode's private data collections and corresponding collection-level endorsement policies. If a collection-level endorsement policy is set, transactions that write to a private data collection key will require that the specified organization peers have endorsed the transaction.

You can use collection-level endorsement policies to restrict which organization peers can write to the private data collection key namespace, for example to ensure that non-authorized organizations cannot write to a collection, and to have confidence that any state in a private data collection has been endorsed by the required collection organization(s).

The collection-level endorsement policy may be less restrictive or more restrictive than the chaincode-level endorsement policy and the collection's private data distribution policy. For example a majority of organizations may be required to endorse a chaincode transaction, but a specific organization may be required to endorse a transaction that includes a key in a specific collection.

The syntax for collection-level endorsement policies exactly matches the syntax for chaincode-level endorsement policies — in the collection configuration you can specify an endorsementPolicy with either a signaturePolicy or channelConfigPolicy.

5. Setting key-level endorsement policies

Setting regular chaincode-level or collection-level endorsement policies is tied to the lifecycle of the corresponding chaincode. They can only be set or modified when defining the chaincode on a channel.

In contrast, key-level endorsement policies can be set and modified in a more granular fashion from within a chaincode. The modification is part of the read-write set of a regular transaction.

The shim API provides the following functions to set and retrieve an endorsement policy for/from a regular key.

```
SetStateValidationParameter(key string, ep []byte) error
GetStateValidationParameter(key string) ([]byte, error)
```

For keys that are part of Private data in a collection the following functions apply:

```
SetPrivateDataValidationParameter(collection, key string, ep []byte) error
GetPrivateDataValidationParameter(collection, key string) ([]byte, error)
```

To help set endorsement policies and marshal them into validation parameter byte arrays, the Go shim provides an extension with convenience functions that allow the chaincode developer to deal with endorsement policies in terms of the MSP identifiers of organizations,

```
type KeyEndorsementPolicy interface {
    // Policy returns the endorsement policy as bytes
    Policy() ([]byte, error)

    // AddOrgs adds the specified orgs to the list of orgs that are required
    // to endorse
    AddOrgs(roleType RoleType, organizations ...string) error

    // DelOrgs delete the specified channel orgs from the existing key-level
endorsement
    // policy for this KVS key. If any org is not present, an error will be
returned.
    DelOrgs(organizations ...string) error

    // ListOrgs returns an array of channel orgs that are required to endorse
changes
    ListOrgs() ([]string)
}
```

For example, to set an endorsement policy for a key where two specific orgs are required to endorse the key change, pass both org MSPIDs to AddOrgs(), and then call Policy() to construct the endorsement policy byte array that can be passed to SetStateValidationParameter().

Validation

At commit time, setting a value of a key is no different from setting the endorsement policy of a key — both update the state of the key and are validated based on the same rules.

Validation	no validation parameter set	validation parameter set
modify value	check chaincode or collection ep	check key-level ep
modify key-level ep	check chaincode or collection ep	check key-level ep

As we discussed above, if a key is modified and no key-level endorsement policy is present, the chaincode-level or collection-level endorsement policy applies by default. This is also true when a key-level endorsement policy is set for a key for the first time — the new key-level endorsement policy must first be endorsed according to the pre-existing chaincode-level or collection-level endorsement policy.

If a key is modified and a key-level endorsement policy is present, the key-level endorsement policy overrides the chaincode-level or collection-level endorsement policy. In practice, this means that the key-level endorsement policy can be either less restrictive or more restrictive than the chaincode-level or collection-level endorsement policies. Because the chaincode-level or collection-level endorsement policy must be satisfied in order to set a key-level endorsement policy for the first time, no trust assumptions have been violated.

If a key's endorsement policy is removed (set to nil), the chaincode-level or collection-level endorsement policy becomes the default again.

If a transaction modifies multiple keys with different associated key-level endorsement policies, all of these policies need to be satisfied in order for the transaction to be valid.