IBM Blockchain Platform

Tutorial B6: Exercising network policies

Estimated time: `20 minutes`

Policies are used to define the rules of how the blockchain network operates. In this tutorial we will:

- Learn about the importance and different kinds of policies
- Exercise the active endorsement policy on DriveNet
- See how smart contracts can implement their own policies

In order to successfully complete this tutorial, you must have first completed tutorial B5: Submitting transactions on the network in your web browser.

There is an optional section at the end that requires two registered users of the DriveNet network to collaborate.

Policies

In the real world, everything we do is subject to rules that determine what we can and can't do. The same applies to the blockchain world: Hyperledger Fabric calls these rules *policies* and are used, among other things, to help maintain privacy and confidentiality, or implement business rules that help reach agreement of transactions.

Policies can be implemented through Hyperledger Fabric configuration options, through smart contract logic, or through business agreements.

However they are implemented, policies are agreed between stakeholders in advance - for example, by the consortium of members or by a regulator. Policies must, of course, fit into the operating environment under which the participants find themselves; a regulator might require visibility of transactions, for example.

Example policies on the DriveNet network include:

- Car IDs must be in the range CAR0-CAR9999. Leading spaces (e.g. CAR0005) are not used.
- Any member of the network can view any car details.
- Only members of IBM Org can update car records CAR0-CAR10.
- Only the current owner of a car record can modify it.
- Updates must be agreed by peers in both the Community Org and the IBM Org.

In this tutorial we'll look in more detail at how the policies work in DriveNet.

`B6.1`: Expand the first section below to get started.

▶ **Endorsement policies**

To begin with, let's look at the following policy:

- Updates to car records must be agreed by peers in both the Community Org and the IBM Org.

Conceptually, this is an example of an *endorsement policy*. Any proposed updates to the world state must be signed (or *endorsed*) by a set of peers that match the active endorsement policy for the smart contract.

When a transaction is submitted to a Hyperledger Fabric blockchain, the transaction is run by all peers addressed by the endorsement policy. Each endorsing peer generates a digitally signed data structure that contains the proposed reads and writes to the world state.

This data structure, including the endorsing peers' signatures, form the transaction content that is compiled into a block by the ordering service. But before each peer applies the updates to its world state, it first checks that the transaction has been correctly signed according to its endorsement policy.

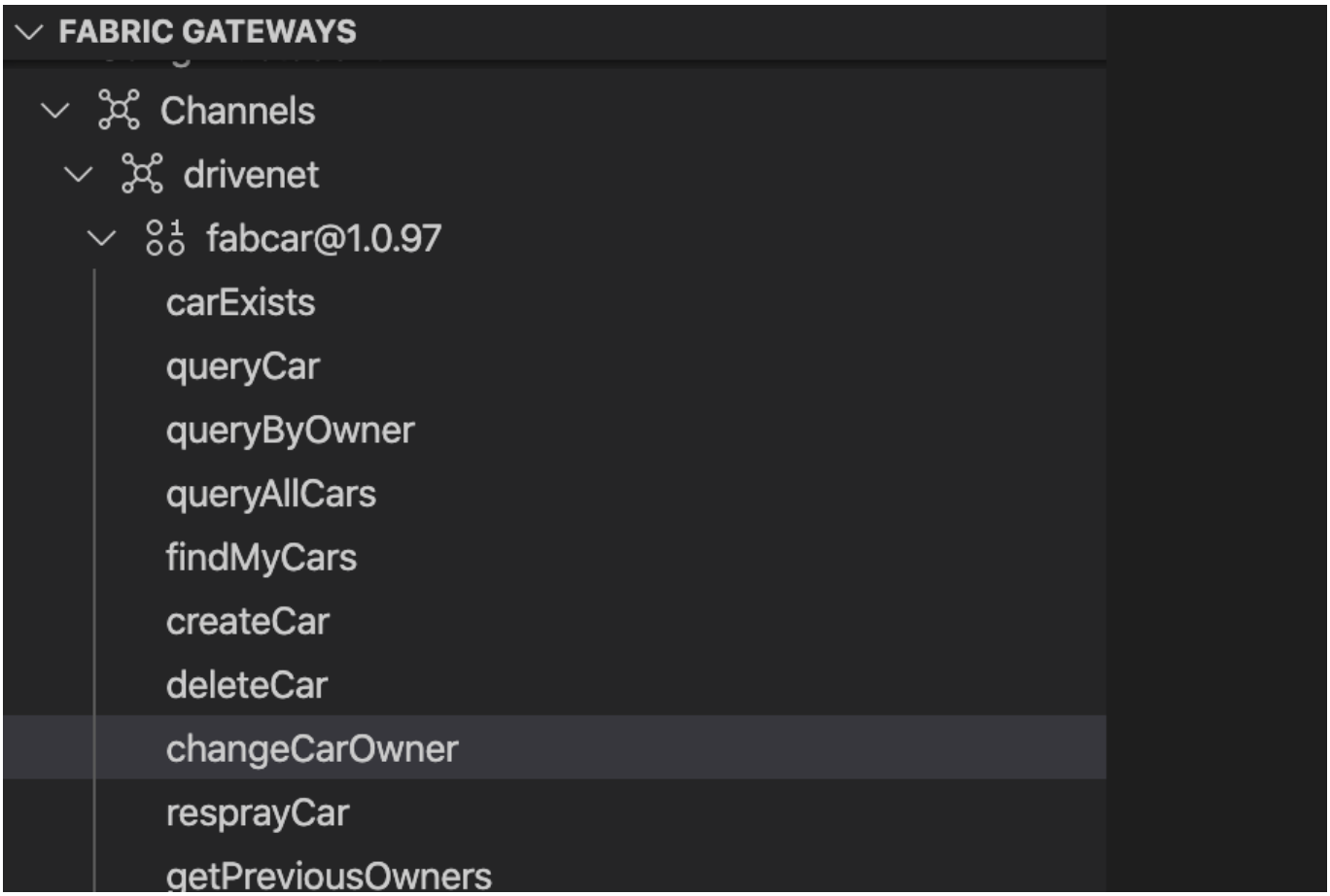
For the fabcar smart contract on DriveNet, each transaction that updates the world state must first be endorsed by both a Community Org peer and an IBM Org peer.

Updating the owner field

Let's test this endorsement policy, and see what happens if we attempt to update a car record without involving peers from both organizations. We'll attempt to submit a transaction to modify the owner field of our asset, but target it *only* at our Community Org peer.

To do this, we'll submit our transaction slightly differently to the way we've done in the past.

 `B6.2`: Switch to the IBM Blockchain Platform VS Code extension, left click the 'changeCarOwner' transaction.



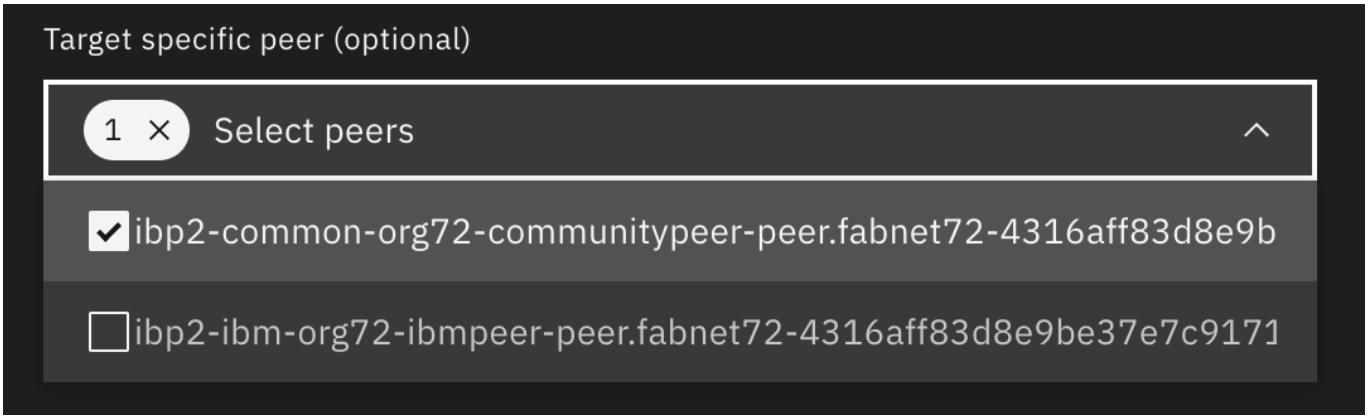
The changeCarOwner transaction takes two arguments: the car ID and the new owner. Use the ID of the car you created in tutorial B5: Invoking transactions on the network, because you won't have the authority to modify other assets. For the new owner, make up a name.

- [ ] B6.3: Update the transaction arguments.

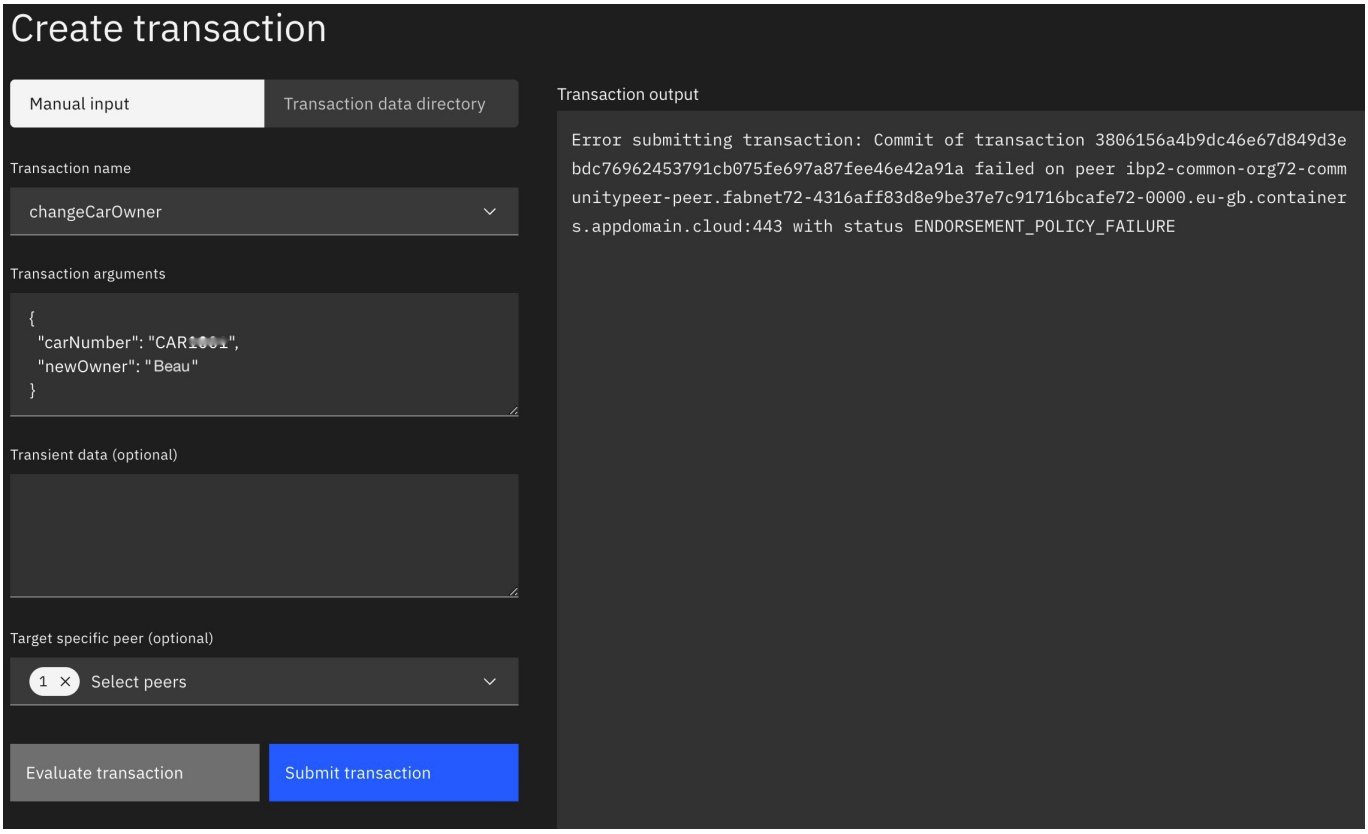
- [ ] B6.4: Ignore the transient data field.

Up until now, we've always specified the default peer targeting policy, which uses the connected gateway to target all endorsing peers. Now we'll only target the Community Org peer.

B6.5: Check only the peer that contains 'communitypeer' in the URL. Leave the URL containing 'ibmpeer' unchecked. Click OK or press Enter to confirm.



The transaction will now only be sent to the Community Org peer for endorsement, and you'll see that it fails with the code *ENDORSEMENT_POLICY_FAILURE*.

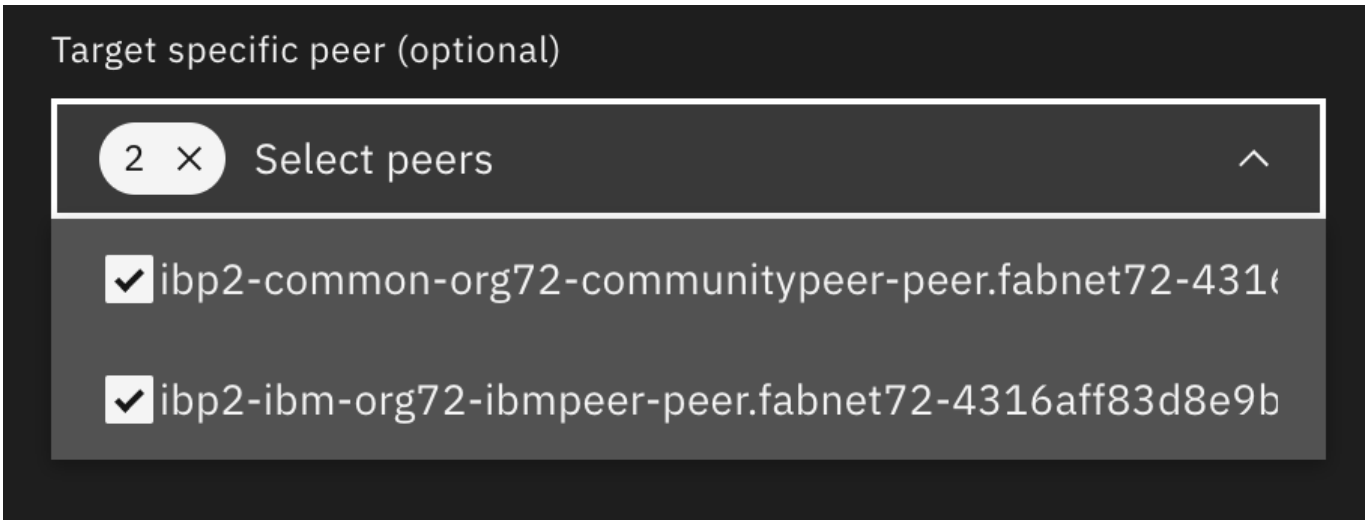


Even though the transaction runs successfully, the peers on the network will refuse to update the world state because the update has not been signed according to its endorsement policy.

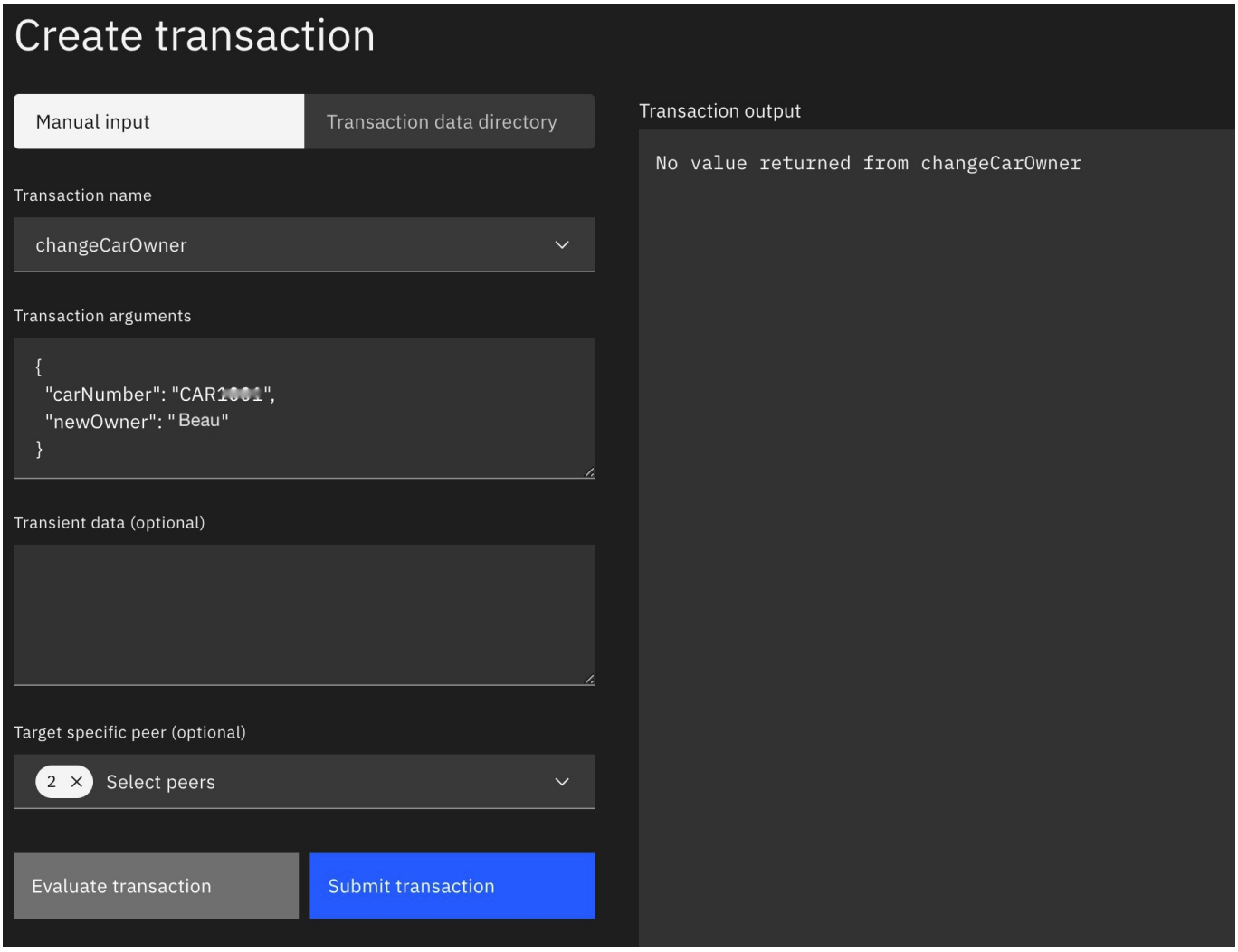
(This transaction has still been added to the blockchain but has been marked as failed. In a later tutorial we'll write an application to demonstrate this.)

For now, let's confirm what happens when we set the endorsement policy correctly.

B6.6: Submit the transaction again; this time, check **both** the IBM Org **and** the Community Org peers before clicking OK.



The transaction now succeeds.

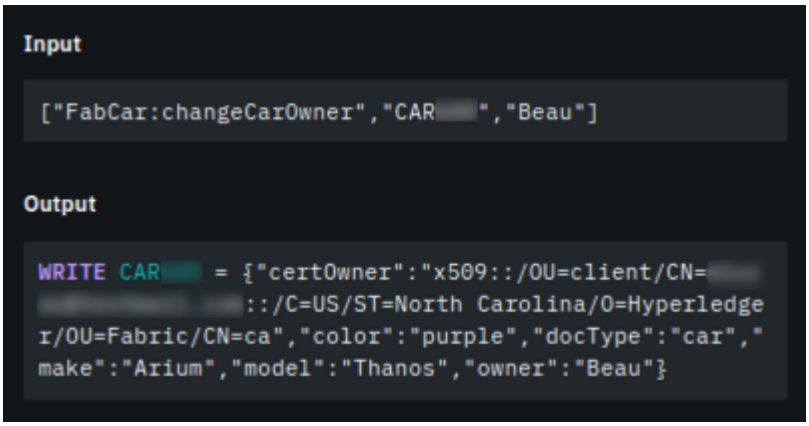


Let's verify that we can see the new transaction in the IBM Blockchain Platform web console.

B6.7: Switch back to the IBM Blockchain Platform web console and click 'drivenet' to return to the DriveNet block history view.

As a result of the transaction you just submitted, there should be a new block.

- B6.8: Navigate into the new block and review the details of your most recent transaction.



You should recognize most of the fields in the WRITE output set. The *certOwner* is used for an advanced scenario; we'll look at this field later on.

- B6.9: Click the Close button to dismiss the side panel.

- B6.10: Expand the next section to continue.

▶ **Policies implemented by the smart contract**

Policies can be also implemented inside the logic of the smart contract. A common pattern is for a smart contract to check the validity of a transaction before updates are made to the world state. If checks fail, an exception is thrown by the smart contract that can be caught by the calling application, and the world state is not updated.

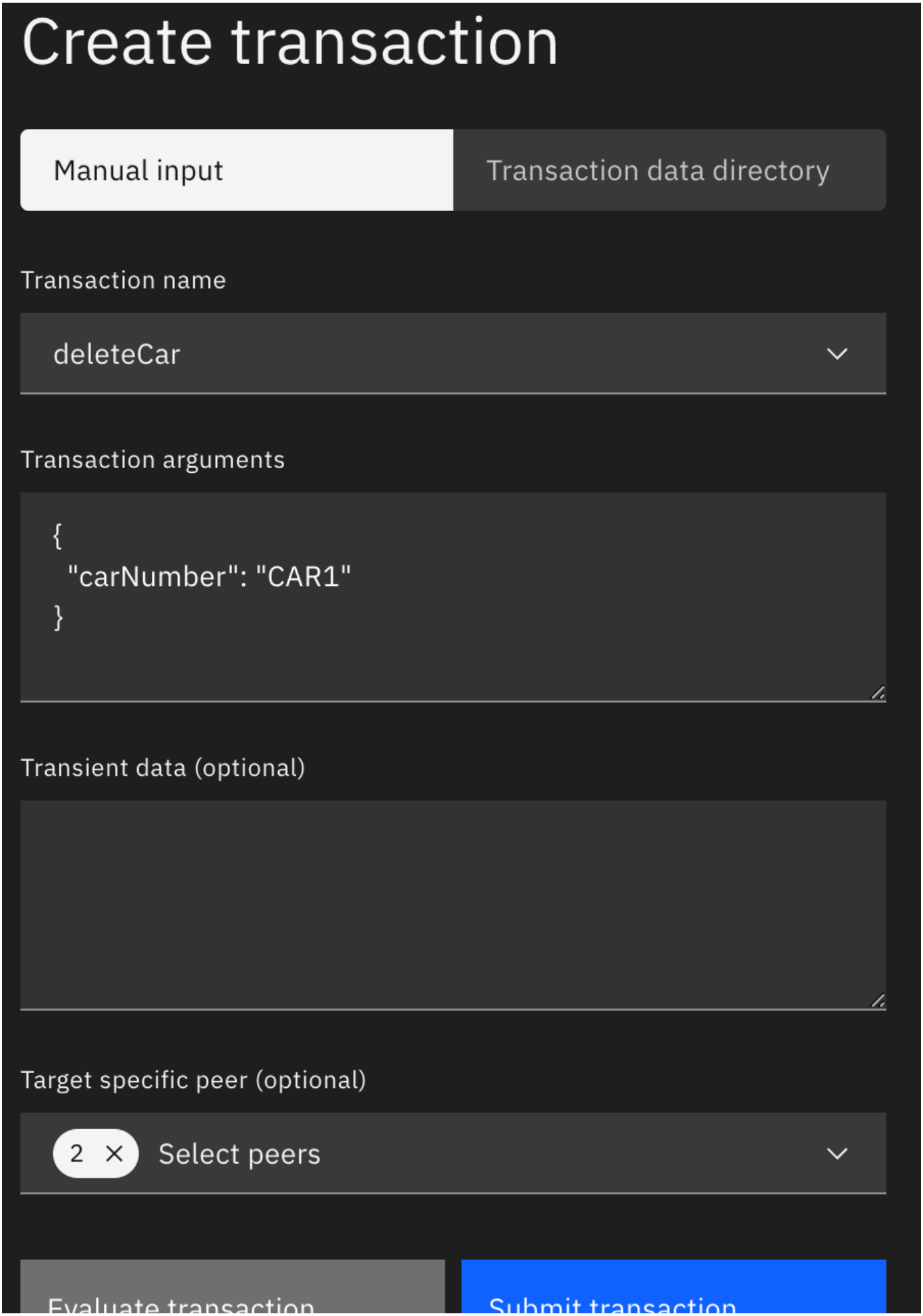
The fabcar smart contract in DriveNet makes several checks before applying state-changing transactions such as *createCar*, *deleteCar* and *changeCarOwner*. For example, whether the user is authorized to modify the car.

In this section we will try some of these policies out.

Attempt to delete an asset

- B6.11: In the IBM Blockchain Platform VS Code extension, submit a 'deleteCar' transaction against the car with ID 'CAR1'.

The deleteCar transaction only takes a single parameter (the car ID). Select the defaults for transient data and peer targeting.



When you submit the transaction, you will see errors in the output window:

Transaction output

```
Error submitting transaction: No valid responses from any peers. Errors:
peer=ibp2-common-org72-communitypeer-peer.fabnet72-4316aff83d8e9be37e7c91716b
cafe72-0000.eu-gb.containers.appdomain.cloud:443, status=500, message=error i
n simulation: transaction returned with failure: Error: The car CAR1 cannot b
e deleted. User fabric_user_4037 not authorised to delete car owned by ibmOrg
Admin.
peer=ibp2-ibm-org72-ibmpeer-peer.fabnet72-4316aff83d8e9be37e7c91716bcafe72-00
00.eu-gb.containers.appdomain.cloud:443, status=500, message=error in simulat
ion: transaction returned with failure: Error: The car CAR1 cannot be delete
d. User fabric_user_4037 not authorised to delete car owned by ibmOrgAdmin.
```

This means that, when running the smart contract code, the IBM and Community peers encountered an error and so did not endorse the transaction. The error was thrown by the smart contract itself; it detected that the submitting organization was not authorized to delete the car.

B6.12: Switch back to the IBM Blockchain Platform web console and browse for your failed transaction.

You'll see that the drivenet channel view does not show the failed transaction.

(Optional) Transfer a car to someone else

> **This final section requires two people**
> Transferring a car requires you to know someone else who is going through these tutorials with you on the same DriveNet instance. If you are working through these tutorials on your own, you can read through the theory but you will not be able to perform the actions; pick things up again with the next tutorial B7: Connecting applications to the network.

As we learned earlier, we can change our car record's owner field to a pretend value and retain the ability to modify it. You should still have write authority to your original car record, despite running the *changeCarOwner* transaction against it earlier.

However, it is possible to hand your write privileges to another registered user, by transferring the ownership of the car to them.

Specifically, the smart contract implements a policy which states that if the current owner changes the owner field to an ID that is registered on the network, then as soon as the new owner confirms the transfer using the *confirmTransfer* transaction, the previous owner will then lose the ability to modify it. In effect, write privileges pass to the new owner.

This two step transfer process ensures that if we set the owner field incorrectly, we have an opportunity to reclaim it. This policy mirrors many real-world transactions: for an asset transfer to be valid, both sender and receiver must agree.

> **Under the hood**
> Write privileges are determined by a hidden field in the car record called '*certOwner*'. The value of this field is the ID of the registered user who can currently write to it.
> You can see the certOwner field when exploring transactions in the web console, but it's hidden from the output of the query transactions. Smart contracts can be selective about the information returned to the caller.

We'll now try out this policy and successfully transfer a car to another registered user, then get them to transfer another car back to us.

- `B6.13`: Share the following information with the other user:

yourFabricID	the Fabric enrollment ID you received via email in tutorial B2: Discovering the network.
yourCarID	the ID of the car you've been working with (CAR*nnn*).
theirFabricID	the Fabric enrollment ID the other user received via email.
theirCarID	the ID of the car the other user has been working with (CAR*mmm*).

- `B6.14`: Ask the other user to attempt to take ownership of your car: They should submit a changeCarOwner transaction with the parameters set to *yourCarID* and *theirFabricID*.

This should fail, because only the current owner of a car can modify it.

- `B6.15`: Transfer your car to the other user: You should submit a changeCarOwner transaction with the parameters *yourCarID*, *theirFabricID*.

They will acquire write privileges to the car record. However, until they confirm ownership of the car, you can also continue to modify it. This is to prevent you from accidentally forfeiting your rights to a record through a typo or pretend value.

- `B6.16`: They should submit a confirmTransfer transaction with the parameters *yourCarID*.

- `B6.17`: Attempt to take back your car. You should submit a changeCarOwner transaction with the parameters set to *theirCarID* and *yourFabricID*.

This will fail, because you are no longer the owner of the car.

- `B6.18`: Reverse roles; let the other user send *theirCarID* to you.

> **Dude, where's my car?**
> In this section you transferred cars as multiple independent transactions. What if you wanted to make the two transactions dependent on each other: *I'll give you my car if (and only if) you give me yours*?
> To do this, the smart contract developer would typically implement a single transaction in the smart contract that performs both updates at the same time. Consensus will ensure that either both updates occur or neither does.
> In the absence of this transaction, remember that the blockchain retains the history of all transactions. At least if your partner absconds with both cars, you have evidence to prove that the deed took place.

Remember the ID of the car that you now own; we'll work with it some more in the next tutorial.

Summary

In this tutorial we looked at the concept of *policies*. Policies can be an intrinsic part of the way a transaction gets agreed by the network - for example, endorsement. Alternatively, they can be implemented programmatically by the smart contract. Either way, they describe the operating rules under which the system works.

So far we have used the IBM Blockchain Platform VS Code extension to submit transactions. This is fine for basic testing, however organizations interacting with production networks will use fully-fledged applications. In the next tutorial we will create a simple client application that connects to DriveNet, and use it to learn more about the assets we've been working with.

→ Next: B7: Connecting applications to the network