Lab 12.2: Setting Custom Attributes

In essence, a user identity represents a X.509 certificate that may have some additional attributes, such as a name and value pairs associated with an identity. We can set different attributes to manage user access to the chaincode logic, ledger entities, and other network resources. For example, we can allow users to invoke the write functions of a chaincode component if and only if they have the `mode` attribute set to `edit`.

You can set any custom attributes during user registration with Fabric CA. `RegisterRequest` has the `attrs` property representing a set of attributes that **can** be added to an enrollment certificate. Each attribute is a JSON object with three properties: `name`, `value`, and `ecert`. While `name` and `value` are self-explanatory, `ecert` is an optional boolean property indicating whether an attribute should be included in the enrollment certificate by default. The default value for `ecert` is `false`, meaning that any attribute is not included in the enrollment certificate by default, and we should somehow choose the attributes that **will** be specified in the certificate.

The list of attributes that will be added to the enrollment certificate can be specified in the `EnrollmentRequest.attr_reqs` field. This field is an array of `AttributeRequest` objects, containing the name of an attribute and the `optional` property. If `optional` is `false`, an exception is thrown in case an identity does not have a specified attribute.

Let's update `registerUser.js` to accept custom attributes. We already use the JSON object containing optional parameters in the code, so let's simply update the `RegisterRequest` object.

```
let registerRequest = {
    enrollmentID: enrollmentID,
    enrollmentSecret: optional.secret || "",
    role: 'client',
    attrs: optional.attrs || []
```

```
};
const secret = await ca.register(registerRequest, registrarUser);
```

Note: There are a number of predefined attributes that can affect permissions of a user in the network. To learn more about this topic, visit the Fabric CA documentation.

We should also update `enrollUser.js` in the same manner. Parse an additional command-line argument and update the `EnrollmentRequest` object as follows.

```
let enrollmentAttributes = [];
if (args.length > 3) {
    enrollmentAttributes = JSON.parse(args[3]);
}

let enrollmentRequest = {
    enrollmentID: enrollmentID,
    enrollmentSecret: enrollmentSecret,
    attr_reqs: enrollmentAttributes
};
const enrollment = await ca.enroll(enrollmentRequest);
```

Now, we can customize certificates of different users and build our business logic based on special attributes. To illustrate a simple workflow based on custom attributes, let's adjust the `BalanceTransfer` chaincode by adding an attribute-based access to the `initAccount` function.

In the very beginning of `BalanceTransfer.initAccount`, check if the transaction creator has the `init` attribute set to `true`. This can be done using the `ClientIdentity` interface discussed in the **Programmatic Access Control: Client Identity** chapter.

```
if (!ctx.clientIdentity.assertAttributeValue("init", "true")) {
    throw new Error(`you don't have permissions to initialize an account`);
}
```

Save the changes and upgrade the chaincode in the network. The complete versions of the `BalanceTransfer` chaincode, `registerUser.js`, and `enrollUser.js` can be found in the attachments.

1) Navigate to the `test-network` folder.

```
# cd $HOME/go/src/github.com/hyperledger/fabric-samples/test-network
```

2) Deploy a new chaincode version. Don't forget to specify the next sequence number.

```
# ./network.sh deployCC -ccn balance_transfer -ccv 2.0 -ccs 2 -ccp
../lfd272/chaincodes/balance_transfer -ccl javascript
```

Next, register and enroll a new user with the `init` attribute set to `true`.

```
# node registerUser.js 'CAAdmin@org1.example.com' 'InitUser@org1.example.com'
'{"secret": "userpw", "attrs": [{"name": "init", "value": "true"}]}'
# node enrollUser.js 'InitUser@org1.example.com' 'InitUser@org1.example.com'
userpw '[{"name": "init", "optional": false}]'
```

Finally, try to submit the `initAccount` transaction on behalf of both `User` and `InitUser`.

```
# node submitTransaction.js 'User@org1.example.com' initAccount acc 100
```

As `User`, you should see an error message notifying you that you cannot perform the account initialization.

```
# node submitTransaction.js 'InitUser@org1.example.com' initAccount acc 100
```

As `InitUser`, you should successfully submit the `initAccount` transaction. You can also list the accounts to make sure that one was created.

```
# node submitTransaction.js 'InitUser@org1.example.com' listAccounts
Response from listAccounts:
[{"balance":100,"id":"acc","owner":"{\"mspid\":\"Org1MSP\",\"id\":\"x509::/OU
=client/CN=InitUser@org1.example.com::/C=US/ST=North
Carolina/L=Durham/O=org1.example.com/CN=ca.org1.example.com\"}"}]
```