Write chaincode

Only the **Retailer** must follow the instructions on this page.

It's now time to write our chaincode. We will be building our chaincode using test-driven development. Begin by replacing `products.js` with the following code. This file is where all the business logic resides. It keeps track of the state of each product in the supply chain and ensures that a product can only transition from one state to another if the appropriate conditions are met. It also verifies that inputs sent to the contract are correct before taking any action.

```
/*
# Copyright 2022 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License.
# A copy of the License is located at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# or in the "license" file accompanying this file. This file is distributed
# on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language governing
# permissions and limitations under the License.
#
*/
'use strict';

const shim = require('fabric-shim');
const log = require('loglevel').getLogger('products');
log.setLevel('trace');
const StateMachine = require('javascript-state-machine');


////////////////////////////////////////////////////////////////////////////
// FSM (Finite State Machine)
// Used to ensure state transitions are valid
////////////////////////////////////////////////////////////////////////////

const FSM = new StateMachine.factory({
  init: 'manufactured',
  transitions: [
    { name: 'inspect', from: 'manufactured', to: 'inspected' }, // supplier
    { name: 'ship', from: 'inspected', to: 'shipped' },         // supplier
    { name: 'receive', from: 'shipped', to: 'stocked' },        // retailer
    { name: 'label', from: 'stocked', to: 'labeled' },          // retailer
    { name: 'sell', from: 'labeled', to: 'sold' },              // retailer
    { name: 'goto', from: '*', to: function(s) { return s } }
```

```
    ]
});

////////////////////////////////////////////////////////////////////////////
// ProductsChaincode
// Used to track all products in the supply chain
////////////////////////////////////////////////////////////////////////////

const ProductsChaincode = class {
  constructor(cid = shim.ClientIdentity) {
    this.clientIdentity = cid;
  }

  ////////////////////////////////////////////////////////////////////////////
  // requireAffiliationAndPermissions
  // Checks that invoke() caller belongs to the specified blockchain member
  // and has the specified permission. Throws an exception if not.
  ////////////////////////////////////////////////////////////////////////////

  requireAffiliationAndPermissions(stub, affiliation, permission) {
    const cid = new this.clientIdentity(stub);
    let permissions = cid.getAttributeValue('permissions') || 'default';
    permissions = permissions.split('_');
    const hasBoth =
      cid.assertAttributeValue('hf.Affiliation', affiliation) &&
      permissions.includes(permission);
    if (!hasBoth) {
      const msg = `Unauthorized access: affiliation ${affiliation}` +
        ` and permission ${permission} required`;
      throw new Error(msg);
    }
  }

  ////////////////////////////////////////////////////////////////////////////
  // assertCanPerformOperation
  // Determines which membership affiliations are required for which
  // operations. Called by other methods. Calls
  // requireAffiliationAndPermissions as a subroutine.
  ////////////////////////////////////////////////////////////////////////////

  assertCanPerformTransition(stub, transition) {
    let requiredAffiliation = 'undefined';
    switch (transition) {
      case 'manufacture':
      case 'inspect':
      case 'ship': requiredAffiliation = 'Supplier'; break;
      case 'receive':
      case 'label':
      case 'sell': requiredAffiliation = 'Retailer';
    }
    this.requireAffiliationAndPermissions(stub, requiredAffiliation, transition);
  }

  ////////////////////////////////////////////////////////////////////////////
  // Initialize the chaincode
```

```
93  ////////////////////////////////////////////////////////////////////////
94
95  async Init(stub) {
96    const ret = stub.getFunctionAndParameters();
97    if (ret.params.length > 0) {
98      return shim.error('Init() does not expect any arguments');
99    }
100   // initialize list of all product IDs so that they can be iterated over
101   await stub.putState('productIDs', Buffer.from('[]'));
102   return shim.success();
103 }
104
105 ////////////////////////////////////////////////////////////////////////
106 // Invoke chaincode and dispatch the appropriate method
107 ////////////////////////////////////////////////////////////////////////
108
109 async Invoke(stub) {
110   const ret = stub.getFunctionAndParameters();
111   log.debug(ret);
112
113   if (!ret.fcn) {
114     return shim.error('Missing method parameter in invoke');
115   }
116
117   let method = this[ret.fcn];
118   let returnval;
119
120   if (!method) {
121     return shim.error(`Unrecognized method ${ret.fcn} in invoke`);
122   }
123   try {
124     let payload = await method(this, stub, ret.params);
125     log.debug(`Payload from call to ${ret.fcn} was ${JSON.stringify(payload)}.`);
126     returnval = shim.success(Buffer.from(payload));
127   } catch (err) {
128     log.error(`Error in Invoke ${ret.fcn}: ${err.message}`);
129     returnval = shim.error(Buffer.from(err.message));
130   }
131   log.debug(`exiting Invoke`)
132   return returnval;
133 }
134
135 ////////////////////////////////////////////////////////////////////////
136 // createProduct
137 // Add a newly-manufactured product to the blockchain
138 ////////////////////////////////////////////////////////////////////////
139
140 async createProduct(self, stub, args) {
141   log.debug(`in createProduct(self, stub, ${JSON.stringify(args)})...`);
142   if (args.length !== 1) {
143     throw new Error('createProduct expects one argument');
144   }
145
146   self.assertCanPerformTransition(stub, 'manufacture');
147   const now = new Date();
```

```
    const payload = {
      "state": "manufactured",
      "history": {
        "manufactured": now.toISOString()
      }
    };
    const strPayload = JSON.stringify(payload);
    const productId = args[0];
    const key = `product_${productId}`;
    let productStateBytes = await stub.getState(key);

    if (!productStateBytes || productStateBytes.length === 0) {
      log.debug(`Calling stub.putState(${key}, Buffer.from(JSON.stringify(${strPayload})))...`);
      await stub.putState(key, Buffer.from(strPayload));
    } else {
      throw new Error('Product with same ID already exists.');
    }

    // add productID to list of product IDs
    log.debug(`Retrieving product ID list...`);
    let arr = await stub.getState('productIDs');
    log.debug(`product ID list, ${arr}`);
    let productIDs = JSON.parse(arr.toString());
    log.debug(`product ID list JSON string, ${productIDs}`);
    productIDs = [...productIDs, key].sort();
    log.debug(`Storing updated product ID list...`);
    await stub.putState('productIDs', Buffer.from(JSON.stringify(productIDs)));

    log.debug(`exiting createProduct(self, stub, ${JSON.stringify(args)})...`);
    return strPayload;
  }

  ////////////////////////////////////////////////////////////////////////
  // updateProductState
  // Update an existing product as it moves through the supply chain
  ////////////////////////////////////////////////////////////////////////

  async updateProductState(self, stub, args) {
    if (args.length !== 2) {
      throw new Error('updateProductState expects two arguments');
    }
    const productId = args[0];
    const transition = args[1];
    self.assertCanPerformTransition(stub, transition);
    const key = `product_${productId}`;
    const productDataBytes = await stub.getState(key);
    const productData = JSON.parse(productDataBytes.toString());
    const product = new FSM();
    product.goto(productData.state);
    product[transition]();
    productData.state = product.state;
    const now = new Date();
    productData.history = productData.history || {};
    productData.history[product.state] = now.toISOString();
    const stringProductData = JSON.stringify(productData);
```

```
    await stub.putState(key, Buffer.from(stringProductData));
    return stringProductData;
  }


  ////////////////////////////////////////////////////////////////////////
  // query blockchain state
  ////////////////////////////////////////////////////////////////////////

  async query() {
    const params = Array.from(arguments);
    let ctx, stub, args, keyIndex = 0, expectedArgLength = 1;
    if (params.length === 2) { // we're being called in unit tests
      [stub, args] = params;
      keyIndex = 1;
      expectedArgLength = 2;
    } else {                   // we're being called in a live environment
      [ctx, stub, args] = params;
    }
    if (args.length !== expectedArgLength) {
      throw new Error(`Incorrect number of arguments. Arguments contains: ${JSON.stringify(args)}`);
    }

    let key = args[keyIndex];

    // Get the state from the ledger
    let resultBytes = await stub.getState(key);
    if (!resultBytes) {
      const message = `No value for key ${key}`;
      throw new Error(message);
    }

    log.debug('Query Response:', resultBytes.toString());
    return resultBytes;
  }
};

module.exports = ProductsChaincode;

if (require.main === module) {
  shim.start(new ProductsChaincode());
}
```

Save that file, then replace `products_test.js` with the following code. This file contains unit tests for the chaincode logic. It ensures that the logic is behaving properly by passing different inputs into the chaincode in a sandboxed environment, and verifying that the expected outputs are given. It is usually best to test your application logic thoroughly before deploying it into production. These tests will also become a useful way of verifying that new application logic hasn't broken any existing features. Developers can run them any time new features are implemented to make sure things continue to behave as expected.

```
/*
# Copyright 2022 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
```

```
4   # Licensed under the Apache License, Version 2.0 (the "License").
5   # You may not use this file except in compliance with the License.
6   # A copy of the License is located at
7   #
8   #     http://www.apache.org/licenses/LICENSE-2.0
9   #
10  # or in the "license" file accompanying this file. This file is distributed
11  # on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
12  # express or implied. See the License for the specific language governing
13  # permissions and limitations under the License.
14  #
15  */
16  'use strict';
17
18  const chai = require('chai');
19  const chaiAsPromised = require('chai-as-promised');
20  const chaiDateTime = require('chai-datetime');
21  chai.use(chaiAsPromised);
22  chai.use(chaiDateTime);
23  const expect = chai.expect;
24  const moment = require('moment');
25
26  const ProductsChaincode = require('./products');
27  const MockStub = require('@theledger/fabric-mock-stub');
28  const { ChaincodeMockStub } = MockStub;
29  const log = require('loglevel').getLogger('products');
30  const log_level = process.env['LOG_LEVEL'] || 'warn';
31  log.setLevel(log_level.toUpperCase());
32
33
34  ////////////////////////////////////////////////////////////////////////////
35  // suppressLogging
36  // helper function to turn off logging during tests
37  // that are supposed to produce errors
38  ////////////////////////////////////////////////////////////////////////////
39
40  const suppressLogging = async (func) => {
41    const previousLevel = log.getLevel();
42    log.setLevel(log.levels.SILENT);
43    await func();
44    log.setLevel(previousLevel);
45  };
46
47
48  ////////////////////////////////////////////////////////////////////////////
49  // spCIDMock
50  // Mock used to imitate the behavior of the ClientIdentity object.
51  ////////////////////////////////////////////////////////////////////////////
52
53  class spCIDMock {
54    constructor(stub) {
55      this._attributes = {
56        'hf.Affiliation': 'Supplier',
57        'permissions': 'manufacture'
58      };
```

```
    }
    getAttributeValue(key) { return this._attributes[key]; }
    assertAttributeValue(key, value) { return this._attributes[key] === value; }
  }


  describe('Products', () => {
    let chaincode, stub;

    beforeEach(() => {
      chaincode = new ProductsChaincode();
      stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
    });

    describe('init', () => {
      it("should succeed", async () => {
        const response = await stub.mockInit('tx1', []);
        expect(response.status).to.eql(200);
      });

      it("should initialize the ProductIDs list", async () => {
        let response = await stub.mockInit('tx1', []);
        response = await chaincode.query(stub, ['query', 'productIDs']);
        expect(response.toString()).to.eql('[]');
      });

      it("should expect no arguments", async () => {
        const response = await stub.mockInit('tx1', ['fn', 'invalid']);
        expect(response.status).to.eql(500);
        expect(response.message).to.eql('Init() does not expect any arguments');
      });
    });

    describe('invoke', () => {
      it('should reject unrecognized commands', async () => {
        const response = await stub.mockInvoke('tx1', ['blah']);
        expect(response.status).to.eql(500);
        expect(response.message).to.eql('Unrecognized method blah in invoke');
      });

      it('should reject invocations with no arguments', async () => {
        const response = await stub.mockInvoke('tx1', []);
        expect(response.status).to.eql(500);
        expect(response.message).to.eql('Missing method parameter in invoke');
      });
    });

    describe('query', () => {
      beforeEach(async () => {
        chaincode = new ProductsChaincode(spCIDMock);
        stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
        await stub.mockInit('tx1', []);
      });

      it('should throw an error if called with too many arguments', async () => {
```

```
114         await expect(chaincode.query(stub, ['query', 'product_1', 'product_2']))
115           .to.eventually.be.rejected.and.match(/Incorrect number of arguments/m);
116       });
117
118       it('should throw an error if a key is not found', async () => {
119         await expect(chaincode.query(stub, ['query', 'product_1']))
120           .to.eventually.be.rejected.and.match(/No value for key/m);
121       });
122
123       it('should return an empty array for productIDs', async () => {
124         const response = await chaincode.query(stub, ['query', 'productIDs']);
125         expect(response.toString()).to.eql('[]');
126       });
127     });
128
129     describe('createProduct', () => {
130       beforeEach(async () => {
131         chaincode = new ProductsChaincode(spCIDMock);
132         stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
133         await stub.mockInit('tx1', []);
134       });
135
136       it("should only accept one argument", async () => {
137         suppressLogging(async () => {
138           await expect(stub.mockInvoke('tx2', ['createProduct', '1', 'extra']))
139             .to.eventually.have.property('message')
140             .and.match(/createProduct expects one argument/m);
141         });
142       });
143
144       it("should not overwrite an existing product with the same ID", async () => {
145         await stub.mockInvoke('tx2', ['createProduct', '1']);
146         suppressLogging(async () => {
147           await expect(stub.mockInvoke('tx3', ['createProduct', '1']))
148             .to.eventually.have.property('message')
149             .and.match(/Product with same ID already exists/m);
150         });
151       });
152
153       it("should set new products' state to 'manufactured'", async () => {
154         let response = await stub.mockInvoke('tx2', ['createProduct', '1']);
155         response = await chaincode.query(stub, ['query', 'product_1']);
156         expect(JSON.parse(response.toString())).to.have.property('state', 'manufactured');
157       });
158
159       it('should add an element to the list of productIDs', async () => {
160         await stub.mockInit('tx2', []);
161         let response = await stub.mockInvoke('tx3', ['createProduct', '1']);
162         response = await chaincode.query(stub, ['query', 'productIDs']);
163         expect(response.toString()).to.equal('["product_1"]');
164       });
165
166       it('should not allow clients without the appropriate permissions', async () => {
167         const productID = '1';
168         chaincode = new ProductsChaincode();
```

```
169       stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
170       await stub.mockInit('tx1', []);
171       async () => {
172         await expect(stub.mockInvoke('tx2', ['createProduct', productID]))
173           .to.eventually.have.property('message')
174           .and.match(/affiliation Supplier and permission manufacture required/m);
175       };
176     });
177   });
178
179   describe('updateProductState', () => {
180     const productID = '1';
181     let chaincode, stub;
182
183     beforeEach(async () => {
184       chaincode = new ProductsChaincode();
185       stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
186     });
187
188     it('should only accept two arguments', async () => {
189       chaincode = new ProductsChaincode(spCIDMock);
190       stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
191       await stub.mockInit('tx1', []);
192       const args = [
193         'updateProductState',
194         productID,
195         'arrivedAtSupplier',
196         'invalid'
197       ];
198       suppressLogging(async () => {
199         await expect(stub.mockInvoke('tx3', args))
200           .to.eventually.have.property('message')
201           .and.match(/updateProductState expects two arguments/m);
202       });
203     });
204
205     it('should not allow invalid state transitions', async () => {
206       class cidMock {
207         constructor(stub) {
208           this._attributes = {
209             'hf.Affiliation': 'Supplier',
210             'permissions': 'manufacture_ship'
211
212           };
213         }
214         getAttributeValue(key) { return this._attributes[key]; }
215         assertAttributeValue(key, value) { return this._attributes[key] === value; }
216       }
217       chaincode = new ProductsChaincode(cidMock);
218       stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
219       await stub.mockInit('tx1', []);
220       const args = ['updateProductState', productID, 'ship'];
221       await stub.mockInvoke('tx2', ['createProduct', productID]);
222       suppressLogging(async () => {
223         let result = await stub.mockInvoke('tx3', args);
```

```
224           expect(result.message.toString()).to.eql('transition is invalid in current state');
225         });
226       });
227
228       it('should store a history of timestamped state transitions', async () => {
229         class cidMock {
230           constructor(stub) {
231             this._attributes = {
232               'hf.Affiliation': 'Supplier',
233               'permissions': 'manufacture_inspect'
234             };
235           }
236           getAttributeValue(key) { return this._attributes[key]; }
237           assertAttributeValue(key, value) { return this._attributes[key] === value; }
238         }
239         chaincode = new ProductsChaincode(cidMock);
240         stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
241         await stub.mockInit('tx1', []);
242         await stub.mockInvoke('tx2', ['createProduct', productID]);
243         await stub.mockInvoke('tx3', ['updateProductState', productID, 'inspect']);
244         const response = await chaincode.query(stub, ['query', 'product_1']);
245         const expectedTime = new Date();
246         const parsedResponse = JSON.parse(response.toString());
247         expect(parsedResponse).to.have.property('history');
248         const time = moment(parsedResponse.history.inspected).toDate();
249         expect(time).to.be.closeToTime(expectedTime, 1);
250       });
251
252       it('should not allow clients without the appropriate permissions', async () => {
253         chaincode = new ProductsChaincode(spCIDMock);
254         stub = new ChaincodeMockStub("ProductsMockStub", chaincode);
255         await stub.mockInit('tx1', []);
256         await stub.mockInvoke('tx2', ['createProduct', productID]);
257         const args = ['updateProductState', productID, 'inspect'];
258         suppressLogging(async () => {
259           let result = await stub.mockInvoke('tx3', args);
260           expect(result.message.toString()).to.match(/affiliation Supplier and permission inspect required/);
261         });
262       });
263     });
264   });
```

Make sure to save these files using **ctrl-s** or **cmd-s**

Verify that the `node_modules` are installed within the chaincode/ directory. If so, run the following command to bundle the dependencies installed in the `node_modules` directory into a local directory `lib/`:

```
1   mv node_modules/ lib
```

Moving the dependencies to `lib/` will allow you to package the installed NPM packages (dependencies) in the chaincode *tar.gz* file in the following steps. Because the node_modules are stored in `lib/`, the Node.js start script in `package.json` has been modified slightly to tell the container environment that runs the chaincode where to find the dependencies at runtime: `"start": "NODE_PATH=lib node products.js"`

At this point, you should be able to run your unit tests and have them pass successfully. Run the following commands from the terminal:

```
nvm use 12.16.1
cd ~/environment/chaincode
npm test
```

You should see the following output:

```
> chaincode@1.0.0 test $HOME/environment/chaincode
> mocha *_test.js



Products
  init
    ✓ should succeed
    ✓ should initialize the ProductIDs list
    ✓ should expect no arguments
  invoke
    ✓ should reject unrecognized commands
    ✓ should reject invocations with no arguments
  query
    ✓ should throw an error if called with too many arguments
    ✓ should throw an error if a key is not found
    ✓ should return an empty array for productIDs
  createProduct
    ✓ should only accept one argument
    ✓ should not overwrite an existing product with the same ID
    ✓ should set new products' state to 'manufactured'
    ✓ should add an element to the list of productIDs
    ✓ should not allow clients without the appropriate permissions
  updateProductState
    ✓ should only accept two arguments
    ✓ should not allow invalid state transitions
    ✓ should store a history of timestamped state transitions
    ✓ should not allow clients without the appropriate permissions


17 passing (36ms)
```

Previous Next