



DALHOUSIE
UNIVERSITY

CSCI 6313 – Introduction to Blockchains

Assignment Number

4

Student Name

Aman Singh Bhandari

Banner ID

B00910008

Gitlab URL

<https://git.cs.dal.ca/bhandari/csci-6313-assignment4>

Table of Contents

<i>Gitlab Repository URL</i>	3
<i>Description of the application</i>	3
Flow of the application.....	3
Smart contract methods.....	3
Setting up IPFS and Microfab.....	7
<i>Dapp</i>	9
<i>Output</i>	21
<i>References</i>	21

Gitlab Repository URL

<https://git.cs.dal.ca/bhandari/csci-6313-assignment4>

This assignment is done with the help of below frameworks, libs, tools and IDE etc.

- IPFS client
- IBM-MicroFab Docker Image
- NodeJS
- Crypto
- web3
- ipfs-http-client
- Javascript

Description of the application

Flow of the application

Figure 1 shows each step taken to solve the notary task for buyer and seller through IPFS and Hyperledger smart contract. The agreement between the buyer and seller are first stored [1] in local IPFS. I am saving the path of the file in the Dapp and reads [1] it for buyer. Buyer then digitally signs it using SHA256 hashing and encrypting through RSA [2] with its private key. I am using crypto package to generate the public-private key pairs and verify the digital signature.

Once buyers signs it, the signature is stored in Hyperledger smart contract through method createAssets008() as shown in figure 3. Same steps are also taken for seller to sign and then store the signature in smart contract through the same method. This method the stores the digital signature of each buyer and seller with key “signature”.

Since now the digital signature sits on blockchain through smart contracts and document sits on IPFS. Next task is to read document and buyer's digital signature and verify it. Smart contract's method readAssets008 () (figure 2) is invoked from Dapp to get the digital signature of buyer. It is then decrypted in Dapp using buyer's public key, which results in a hash. The plain text retrieved from IPFS is then hashed using the same algo – SHA256. Both the hashes are compared to verify the authenticity of document and buyer [2]. Same steps are repeated for seller as well.

Smart contract methods

- createAssets008()– Stores the digital signature of the buyer and seller. The digital signature is constructed in Dapp and set through this method of smart contract.
- readAssets008()– Gets the digital signature of the buyer. The digital signature is constructed in Dapp and get through this method of smart contract.

- `assetExists008()` – Checks and confirm whether the asset is present in the blockchain with given ID. It is created to avoid creating the objects with the same id again.
- `approveOrCancelAgreement008()`– Approves or rejects the document by changing the parameter called “state” in the asset.
- `updateAssets008()`- Updates the states.
- `deleteAssets008()`- Deletes the state.

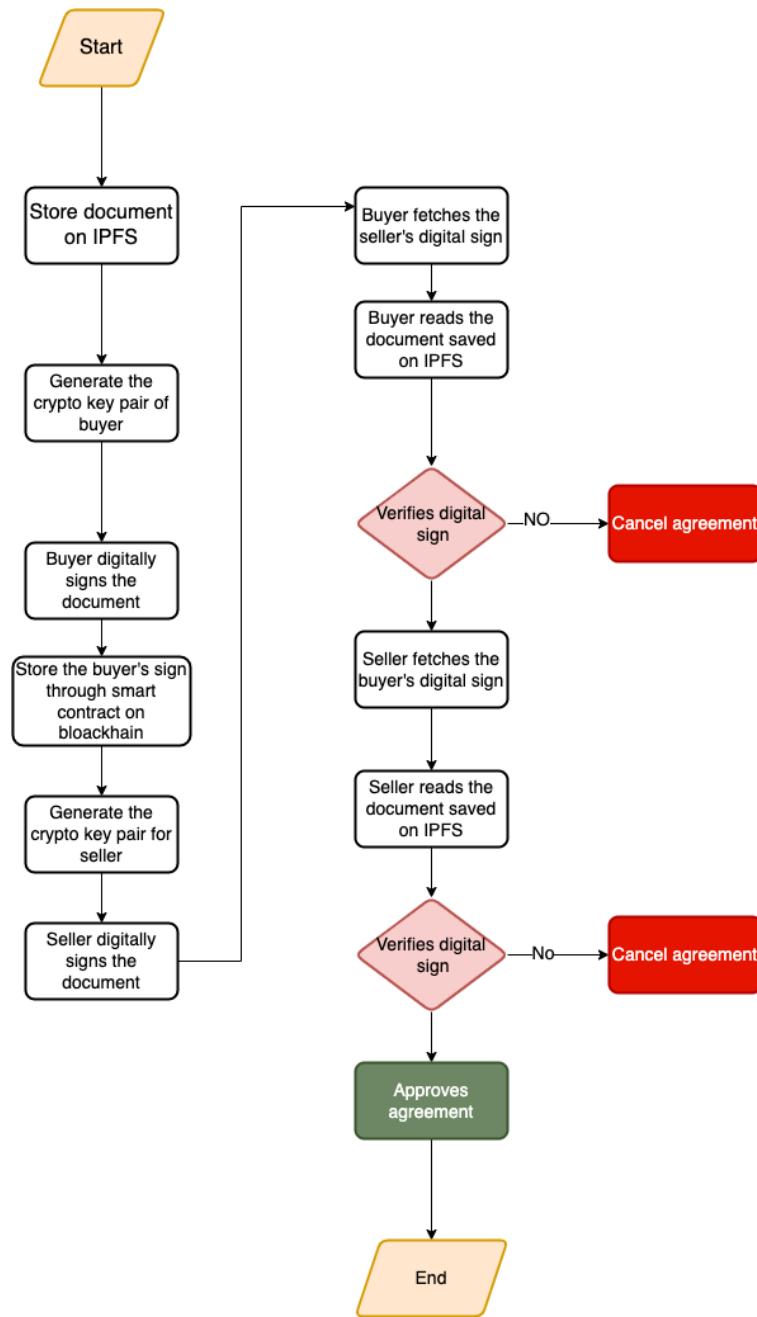


Figure 1 flow diagram

notary-smart-contract008.js — smart-contract

```

JS notary-smart-contract008.js ×

lib > JS notary-smart-contract008.js > NotarySmartContract008
1  /*
2   * SPDX-License-Identifier: Apache-2.0
3   */
4
5 "use strict";
6
7 const { Contract } = require("fabric-contract-api");
8 const crypto = require("crypto");
9
10 class NotarySmartContract008 extends Contract {
11   /**
12    *
13    * @param {*} ctx context
14    * @param {*} deliverableAssetsId key if the buyer or seller signature
15    * @returns if the buyer's or seller's object exists or not
16    */
17   async assetExists008(ctx, deliverableAssetsId) {
18     const buffer = await ctx.stub.getState(deliverableAssetsId);
19     return !!buffer && buffer.length > 0;
20   }
21
22   /**
23    *
24    * @param {*} ctx context
25    * @param {*} deliverableAssetsId key if the buyer or seller signature
26    * @param {*} value status of the agreement accept or rejected
27    */
28   async approveOrCancelAgreement008(ctx, deliverableAssetsId, value) {
29     const exists = await this.assetExists008(ctx, deliverableAssetsId);
30
31     if (!exists) {
32       throw new Error(`The agreement ${deliverableAssetsId} does not exist`);
33     }
34
35     const asset = await this.readAssets008(ctx, deliverableAssetsId); //get asset
36     asset.status = value; //change the status to approved
37     const buffer = Buffer.from(JSON.stringify(asset)); //stringify it
38     await ctx.stub.putState(deliverableAssetsId, buffer); //update it
39   }
40
41   /**
42    *
43    * @param {*} ctx context
44    * @param {*} deliverableAssetsId key of the sign of either buyer or seller
45    * @param {*} value signature of buyer or seller
46    */

```

Figure 2 Hyperledger smart contract

notary-smart-contract008.js — smart-contract

```

JS notary-smart-contract008.js ×

lib > JS notary-smart-contract008.js > NotarySmartContract008
47     async createAssets008(ctx, deliverableAssetsId, value) {
48         const exists = await this.assetExists008(ctx, deliverableAssetsId);
49
50         if (exists) {
51             throw new Error(`The agreement ${deliverableAssetsId} already exists`);
52         }
53         const status = "";
54         const asset = { signature: value, status };
55         const buffer = Buffer.from(JSON.stringify(asset));
56         await ctx.stub.putState(deliverableAssetsId, buffer);
57     }
58
59 /**
60 *
61 * @param {*} ctx context
62 * @param {*} deliverableAssetsId key if the buyer or seller signature
63 * @returns object of the signature of buyer or seller
64 */
65 async readAssets008(ctx, deliverableAssetsId) {
66     const exists = await this.assetExists008(ctx, deliverableAssetsId);
67     if (!exists) {
68         throw new Error(`The agreement ${deliverableAssetsId} does not exist`);
69     }
70     const buffer = await ctx.stub.getState(deliverableAssetsId);
71     const asset = JSON.parse(buffer.toString());
72     return asset;
73 }
74
75 /**
76 *
77 * @param {*} ctx context
78 * @param {*} deliverableAssetsId key if the buyer or seller signature
79 * @param {*} newValue new value of digital signature
80 */
81 async updateAssets008(ctx, deliverableAssetsId, newValue) {
82     const exists = await this.assetExists008(ctx, deliverableAssetsId);
83     if (!exists) {
84         throw new Error(`The agreement ${deliverableAssetsId} does not exist`);
85     }
86     const status = "";
87     const asset = { agreement: newValue, status };
88     const buffer = Buffer.from(JSON.stringify(asset));
89     await ctx.stub.putState(deliverableAssetsId, buffer);
90 }
91
92 /**
93

```

Figure 3 Hyperledger smart contract cont..

```

JS notary-smart-contract008.js ×
lib > JS notary-smart-contract008.js > NotarySmartContract008
15  /**
16   * @param {*} ctx context
17   * @param {*} deliverableAssetsId key if the buyer or seller signature
18   * @param {*} newValue new value of digital signature
19   */
20
21  async updateAssets008(ctx, deliverableAssetsId, newValue) {
22    const exists = await this.assetExists008(ctx, deliverableAssetsId);
23    if (!exists) {
24      throw new Error(`The agreement ${deliverableAssetsId} does not exist`);
25    }
26    const status = "";
27    const asset = { agreement: newValue, status };
28    const buffer = Buffer.from(JSON.stringify(asset));
29    await ctx.stub.putState(deliverableAssetsId, buffer);
30  }
31
32 /**
33 *
34 * @param {*} ctx context
35 * @param {*} deliverableAssetsId key if the buyer or seller signature
36 */
37
38 async deleteAssets008(ctx, deliverableAssetsId) {
39   const exists = await this.assetExists008(ctx, deliverableAssetsId);
40   if (!exists) {
41     throw new Error(`The agreement ${deliverableAssetsId} does not exist`);
42   }
43   await ctx.stub.deleteState(deliverableAssetsId);
44 }
45
46 module.exports = NotarySmartContract008;
47

```

Figure 4 Hyperledger smart contract cont..

Setting up IPFS and Microfab

Created and tested a smart contract in javascript using visual code extension called IBM Blockchain Platform. Once the smart contract is deployed, I tested the same with few dummy values to check if it is working or not. Installed the local IPFS in machine and started it using “ipfs daemon”. Downloaded the ibmcom/ibp-microfab docker image and ran it. Please check the local IPFS running on figure 5 and ibmcom/ibp-microfab container logs in figure 6. Figure 7 shows the agreement between buyer and seller on local IPFS.

```

...com.docker.cli ~ docker run -p 8080:8080 ibmcom/ibp-microfab ~ -- ipfs daemon
Amans-MacBook-Pro-2:~ amansinghbhandari$ ipfs daemon
Initializing daemon...
go-ipfs version: 0.13.1
Repo version: 12
System version: amd64/darwin
Golang version: go1.18.3
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/127.0.0.1/udp/4001/quic
Swarm listening on /ip4/192.168.0.3/tcp/4001
Swarm listening on /ip4/192.168.0.3/udp/4001/quic
Swarm listening on /ip6::1/tcp/4001
Swarm listening on /ip6::1/udp/4001/quic
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/127.0.0.1/udp/4001/quic
Swarm announcing /ip4/192.168.0.3/tcp/4001
Swarm announcing /ip4/192.168.0.3/udp/4001/quic
Swarm announcing /ip4/38.21.184.54/udp/4001/quic
Swarm announcing /ip6::1/tcp/4001
Swarm announcing /ip6::1/udp/4001/quic
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8082
Daemon is ready
2022-07-25T02:42:30.732-0300    ERROR    tcp-tpt go-tcp-transport@v0.5.1/tcp.go:59      failed
    set keepalive period      {"error": "set tcp4 192.168.0.2:4001->3.36.93.174:4001: setsockopt: invalid argument"}
2022-07-25T02:47:59.338-0300    ERROR    tcp-tpt go-tcp-transport@v0.5.1/tcp.go:59      failed
    set keepalive period      {"error": "set tcp4 192.168.0.2:4001->67.177.115.182:44005: setsockopt: invalid argument"}

```

Figure 5 local IPFS

```

...com.docker.cli ~ docker run -p 8080:8080 ibmcom/ibp-microfab ~ -- ipfs daemon
Q~ 8080 ~ -- ipfs daemon
[ org1peer] 2022-07-25 05:44:36.506 UTC 0162 INFO [gossip.privdata] StoreBlock -> Received block [7] from buffer channel=channel1
[ org1peer] 2022-07-25 05:44:36.507 UTC 0163 INFO [committer.txvalidator] Validate -> [channel1] Validated block [7] in 1ms
[ org1peer] 2022-07-25 05:44:36.549 UTC 0164 INFO [kvledger] commit -> [channel1] Committed block [7] with 1 transaction(s) in 39ms (state_validation=3ms block_and_pvtdata_commit=6ms state_commit=27ms) commitHash=[c2523957b0797bed0cd8d4677d539b44b2228212a5677ef908f273310c892476]
[ org1peer] 2022-07-25 05:44:36.709 UTC 0165 INFO [chaincode.externalbuilder.node] waitForExit -> 2022-07-25T05:44:36.700Z info [c-api:lib/handler.js] [channel1-3ec29d94] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer command=run
[ org1peer] 2022-07-25 05:44:36.709 UTC 0166 INFO [endorser] callChaincode -> finished chaincode: assignment2 duration: 83ms channel=channel1 txID=3ec29d94
[ org1peer] 2022-07-25 05:44:36.711 UTC 0167 INFO [comm.grpc.server] 1 -> unary call completed grpc.service=protos.Endorser grpc.method=ProcessProposal grpc.peer_address=127.0.0.1:37356 grpc.code=OK grpc.call_duration=86.731432ms
[ org1peer] 2022-07-25 05:44:36.872 UTC 0168 INFO [chaincode.externalbuilder.node] waitForExit -> 2022-07-25T05:44:36.841Z info [c-api:lib/handler.js] [channel1-d7e486b2] Calling chaincode Invoke() succeeded. Sending COMPLETED message back to peer command=run
[ org1peer] 2022-07-25 05:44:36.887 UTC 0169 INFO [endorser] callChaincode -> finished chaincode: assignment2 duration: 112ms channel=channel1 txID=d7e486b2
[ org1peer] 2022-07-25 05:44:36.887 UTC 016a INFO [comm.grpc.server] 1 -> unary call completed grpc.service=protos.Endorser grpc.method=ProcessProposal grpc.peer_address=127.0.0.1:37356 grpc.code=OK grpc.call_duration=116.224342ms

```

Figure 6 ibm-microfab docker container

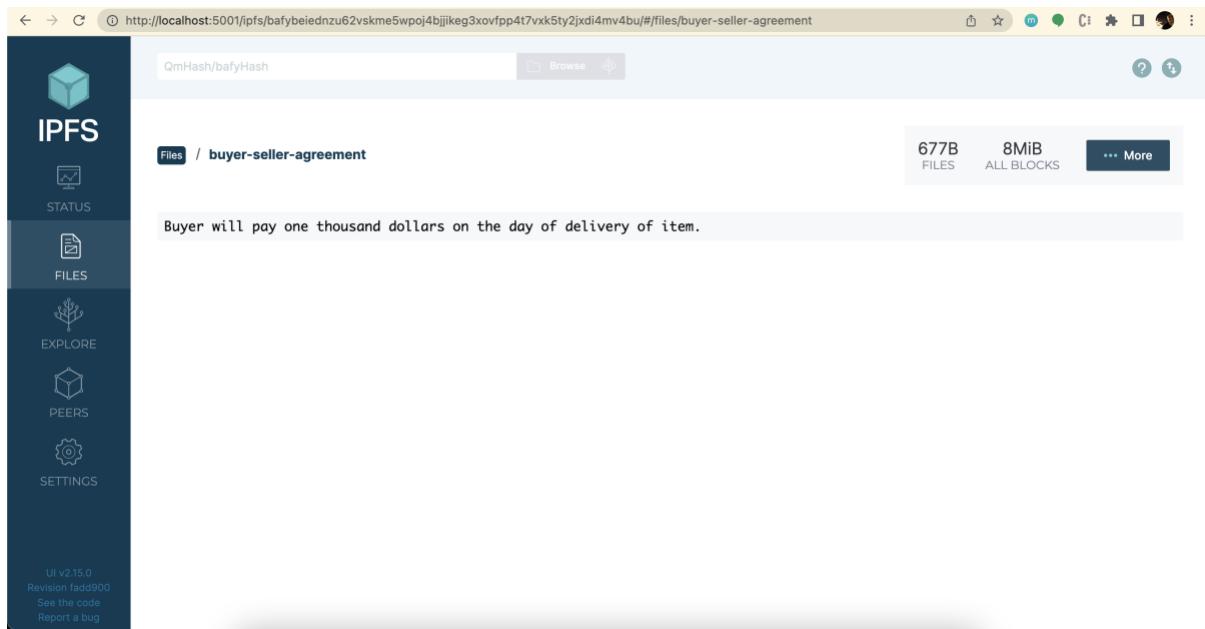


Figure 7 IPFS webui

Dapp

Dapp is built with NodeJS. It can be run through the command “node index.js”. I replaced the Org1 (wallet) folder and gateway JSON file. Dapp will run with one single command and all the steps are performed sequentially and is displayed on the console. The Dapp is a basic application that doesn’t have any UI but outputs all logs in the console.

Please check the next section to see the output.

Below are the screenshots of the distributed application that interact with blockchain through Hyperledger smart contract and IPFS.

```
JS index.js  X

JS index.js > [ø] requestListener
1  /**
2   * @author aman singh bhandari
3   * @bannerid B00910008
4   */
5  import { createRequire } from "module";
6  import { create } from "ipfs-http-client";
7  import { AbortController } from "node-abort-controller";
8  import { fileURLToPath } from "url";
9
10 const require = createRequire(import.meta.url);
11
12 var Web3 = require("web3");
13 const crypto = require("crypto");
14 const client = create();
15 var MFS_path008 = "/buyer-seller-agreement";
16 global.AbortController = AbortController;
17 let details = "{price:12ETH, item:rent-bmw3series}"; //Agreement between parties
18
19 const { Gateway, Wallets } = require("fabric-network");
20 const path = require("path");
21
22 const AppUtil = require("./lib/AppUtil.cjs");
23
24 const { buildWallet } = require("./lib/AppUtil.cjs");
25
26 const __filename = fileURLToPath(import.meta.url);
27 const __dirname = path.dirname(__filename);
28 const walletPath = path.join(__dirname, "Org1"); //Org1 is exported and is kept in root folder
29
30 const http = require("http"); //protocol
31 const url = require("url");
32
33 const host = "0.0.0.0"; //localhost
34 const port = 8083; //port to listen
35
36 let identity = "Org1 Admin";
37 let networkConnections = {};
38 let gateway = null;
```

```
JS index.js  X

JS index.js > [ø] requestListener
38   let gateway = null;
39   let network = null;
40   let contract = null;
41
42   const buyerKeyPair008 = crypto.generateKeyPairSync("rsa", {
43     modulusLength: 2048,
44     publicKeyEncoding: {
45       type: "spki",
46       format: "der",
47     },
48     privateKeyEncoding: {
49       type: "pkcs8",
50       format: "der",
51     },
52   });
53
54   const sellerKeyPair008 = crypto.generateKeyPairSync("rsa", {
55     modulusLength: 2048,
56     publicKeyEncoding: {
57       type: "spki",
58       format: "der",
59     },
60     privateKeyEncoding: {
61       type: "pkcs8",
62       format: "der",
63     },
64   });
65
66   async function initNetwork008() {
67     try {
68       const ccp = AppUtil.buildJunglekidsOrg1();
69       const wallet = await buildWallet(Wallets, walletPath);
70       if (gateway == null) gateway = new Gateway();
71
72       if (network == null) {
73         console.log("\nNetwork is not created yet.");
74         console.log("Build a network gateway to connect to local emulator");
75     }
76   }
77 }
```

```
JS index.js  X
JS index.js > [?] requestListener
● 75      await gateway.connect(ccp, {
76        wallet,
77        identity: identity, //..json exported from contract
78        discovery: { enabled: true, aslocalhost: false }, //using emulator
79      });
80    }
81  }
82
83  await submitAgreement008(); //write the agreement between buyer and seller in IPFS.
84  await fetchAndSignDocBuyer008(); //Buyer will fetch and sign the document
85  await fetchAndSignDocSeller008(); //Seller will fetch and sign the document
86  await verifyBuyerDigiSign008(); //Verify Buyer's digital signature with buyer's public key
87  await verifySellerDigiSign008(); //Verify Seller's digital signature with Seller's public key
88 } catch (error) {
89   console.error(`initializeHyperledgerNetwork error: ${error}`);
90 }
91 }
92
93 var fetchAndSignDocBuyer008 = async function () {
94   console.log("\n##### Buyer Signing document #####");
95   const raw = await readDocumentFromIPFS008();
96   console.log(
97     "Buyer read the document from IPFS with content:: " + ' ' + raw + ' '
98   );
99
100  const privateKey = crypto.createPrivateKey({
101    key: buyerKeyPair008.privateKey,
102    type: "pkcs8",
103    format: "der",
104  });
105  console.log("Buyer will sign the document now..");
106  //---Sign document ---
107  const sign = crypto.createSign("SHA256");
108  console.log("Hashing SHA256..");
109  sign.update(raw);
110  sign.end();
111  const signature = sign.sign(privateKey).toString("base64");
112  console.log("Encrypting the hash with buyer's private key..");
```

```
JS index.js  X

JS index.js > [ø] requestListener
112     console.log("Encrypting the hash with buyer's private key..");
113     console.log("Signature of buyer completed successfully!");
114     console.log("Digital signature is ", signature);
115
116     await createAsset("BuyerSignature", signature);
117 };
118
119 var fetchAndSignDocSeller008 = async function () {
120     console.log("\n##### Seller Signing document #####");
121     const raw = await readDocumentFromIPFS008();
122     console.log(
123         "Seller read the document from IPFS with content:: " + '""' + raw + '""'
124     );
125
126     const privateKey = crypto.createPrivateKey({
127         key: sellerKeyPair008.privateKey,
128         type: "pkcs8",
129         format: "der",
130     });
131     console.log("Seller will sign the document now..");
132     //---Sign document ---
133     const sign = crypto.createSign("SHA256");
134     console.log("Hashing SHA256..");
135     sign.update(raw);
136     sign.end();
137     const signature = sign.sign(privateKey).toString("base64");
138     console.log("Encrypting the hash with seller's private key..");
139     console.log("Signature of seller completed successfully!");
140     console.log("Digital signature is ", signature);
141
142     await createAsset("SellerSignature", signature);
143 };
144
145 var verifyBuyerDigiSign008 = async function () {
146     console.log("\n##### Verify buyer's signatures #####");
147
148     const raw = await readDocumentFromIPFS008();
149     console.log("Read plain text of document from IPFS..");
```

```
JS index.js  X

JS index.js > [ø] requestListener
150 const result = await readAsset("BuyerSignature");
151 let resultjson = JSON.parse(result);
152 const value = resultjson.signature;
153 console.log("Fetch buyer's signature from smart contract (getter method)..");
154 console.log("Seller will now verify the digital sign of Buyer..");
155 const sign = Buffer.from(value, "base64");
156 const publicKey = crypto.createPublicKey({
157   key: buyerKeyPair008.publicKey,
158   type: "spki",
159   format: "der",
160 });
161 console.log("Using public key of buyer..");

162 const verify = crypto.createVerify("SHA256");
163 verify.update(raw);
164 verify.end();
165 console.log("Hashing the plain text gotten from IPFS..");
166 let isVerified = verify.verify(publicKey, sign);
167 console.log("Digital sign of buyer verified with result:: ", isVerified);
168 };
169
170 var verifySellerDigiSign008 = async function () {
171   console.log("\n##### Verify seller's signatures #####");
172
173   const raw = await readDocumentFromIPFS008();
174   console.log("Read plain text of document from IPFS..");
175   const result = await readAsset("SellerSignature");
176   let resultjson = JSON.parse(result);
177   const value = resultjson.signature;
178   console.log("Fetch seller's signature from smart contract (getter method)..");
179   console.log("Buyer will now verify the digital sign of Seller..");
180   const sign = Buffer.from(value, "base64");
181   const publicKey = crypto.createPublicKey({
182     key: sellerKeyPair008.publicKey,
183     type: "spki",
184     format: "der",
185   });
186   console.log("Using public key of seller..").
```

```
JS index.js  X

JS index.js > [ø] requestListener
187    | console.log("Using public key of seller..");
188
189    const verify = crypto.createVerify("SHA256");
190    verify.update(raw);
191    verify.end();
192    console.log("Hashing the plain text gotten from IPFS..");
193    let isVerified = verify.verify(publicKey, sign);
194    console.log("Digital sign of seller verified with result:: ", isVerified);
195
196    console.log("\nThe signature of both buyer and seller are vaerified..");
197    console.log("Hurrrayy !! Approving the agreement!!");
198  };
199
200 var readDocumentFromIPFS008 = async function () {
201  return client.files.stat(MFS_path008, { hash: true }).then(async (r) => {
202    let ipfsAddr008 = r.cid.toString();
203    //reading the content
204    const resp = await client.cat(ipfsAddr008);
205    let content = [];
206    for await (const chunk of resp) {
207      content = [...content, ...chunk];
208      const raw = Buffer.from(content).toString("utf8");
209      return raw;
210    }
211  });
212};
213
214 var submitAgreement008 = async function () {
215  global.AbortController = AbortController;
216  const agreement008 =
217    "Buyer will pay one thousand dollars on the day of delivery of item."; //Agreement between buyer
218  console.log(
219    "\nSaving the agreement between buyer and seller in local IPFS: " +
220    agreement008
221  );
222
223  return client.files
224    .write(MFS_path008, new TextEncoder().encode(agreement008), {
225      signal: global.AbortController.signal
    }
```

```
JS index.js  ×

JS index.js > [ø] requestListener
224  |   .write(MFS_path008, new TextEncoder().encode(agreement008), {
225  |     create: true,
226  |   })
227  |   .then(async (r) => {
228  |     client.files.stat(MFS_path008, { hash: true }).then(async (r) => {
229  |       let ipfsAddr008 = r.cid.toString();
230  |       console.log("Stored file on IPFS with address ", ipfsAddr008);
231  |     });
232  |   })
233  |   .catch((e) => {
234  |     console.log(e);
235  |   });
236  };
237
238 < async function initContract() {
239  |   try {
240  |     console.log(
241  |       "\nFrom the gateway created to access the emulator, retreive the channel"
242  |     );
243  |     network = await gateway.getNetwork("channel1"); //get the channel
244  |     console.log("Got the network gateway of specified channel");
245  |     contract = network.getContract("assignment2"); //get contract to access its method later
246  |     console.log("Got the contract");
247  |     networkConnections["assignment2"] = contract; //add to connection map
248  |     return contract;
249  |   } catch (error) {
250  |     console.error(`\nContract initialization error: ${error}`);
251  |   }
252  }
253
254 < /**
255  *
256  * verifies if the contract is loaded and
257  */
258 < async function getActorConnection() {
259  |   if (!networkConnections["assignment2"]) {
260  |     await initContract();
261  |   }

```

```
JS index.js  X
JS index.js > [ø] requestListener
261     }
262     return networkConnections["assignment2"];
263   }
264
265   ∵ async function approveOrCancelAgreement(id, value) {
266     let contract = await getActorConnection();
267     let result = "";
268
269     ∵ try {
270       await contract.submitTransaction("approveOrCancelAgreement008", id, value); //approve or reject
271       result = "Agreement with Id " + id + " was successfully submitted!";
272     } catch (e) {
273       result = e.message;
274     }
275     console.log("\n" + result);
276     return result;
277   }
278
279   ∵ async function createAsset(id, value) {
280     let contract = await getActorConnection();
281     let result = "";
282     console.log("Creating asset with id = " + id + ", value = " + value);
283
284     ∵ try {
285       await contract.submitTransaction("createAssets008", id, value); //submit agreement
286       result = "Agreement with Id " + id + " was successfully submitted!";
287     } catch (e) {
288       result = e.message;
289     }
290     console.log("\n" + result);
291     return result;
292   }
293
294   ∵ async function readAsset(id) {
295     ∵ console.log(
296       "\n---- Now Buyer/Seller wants to retreive the agreement to validate it -----"
297     );
298     let contract = await getActorConnection();
```

```
JS index.js  X

JS index.js > [ø] requestListener
297  ,
298  let contract = await getActorConnection();
299  let result = "";
300  try {
301    result = await contract.evaluateTransaction("readAssets008", id);
302  } catch (e) {
303    result = e.message;
304  }
305  console.log(
306    "On retrieving the Buyer/Seller got the below agreement and supporting fields \n" +
307    |   result
308  );
309  return result;
310}
311
312const requestListener = async function (req, res) {
313  const queryObject = url.parse(req.url, true).query;
314
315  console.log("req.url:", req.url);
316
317  let result = "";
318  let id = "";
319  let value = "";
320  let status = "";
321  const crypto = require("crypto");
322
323  res.setHeader("Content-Type", "application/json");
324
325  if (req.url.startsWith("/retrieve-agreement")) {
326    id = queryObject.id;
327    result = await readAsset(id);
328
329    let resultjson = JSON.parse(result);
330    const agreement = resultjson.agreement;
331
332    const retrievedHash = resultjson.hash;
333    console.log("\nretrieved agreement from the contract: " + agreement);
334    console.log("retrieved hash from the contract: " + retrievedHash);
335  }
336}
```

```
JS index.js  X

JS index.js > [0] requestListener
333 |     console.log(`... retrieved agreement from the contract: ${agreement}`);
334 |     console.log("retrieved hash from the contract: " + retrievedHash);
335 |     console.log(
336 |         "\nBuyer/Seller will now calculate hash of retrieved agreement (md5): "
337 |     );
338 |
339 |     var calculatedHash = crypto
340 |         .createHash("md5")
341 |         .update(agreement)
342 |         .digest("hex"); //Hashing with md5
343 |
344 |     console.log("Calculated hash is: " + calculatedHash);
345 |
346 |     if (calculatedHash === retrievedHash) {
347 |         console.log(
348 |             "\nYayy !!! Hash matched that means the agreement integrity is verified!"
349 |         );
350 |         console.log("Buyer/Seller will now APPROVE the agreement");
351 |         resultjson.message =
352 |             "Buyer retrived the asset -> calculated its hash -> It matched :) !!!";
353 |     } else {
354 |         console.log("\nOppsss !!! Hash didn't match");
355 |         console.log("Buyer/Seller will now CANCEL the agreement");
356 |         resultjson.message =
357 |             "Buyer retrived the asset -> calculated its hash -> And it didn't match :( !!!";
358 |     }
359 |     result = JSON.stringify(resultjson);
360 |     res.writeHead(200);
361 |     res.end(result);
362 | } else if (req.url.startsWith("/approve-agreement")) {
363 |     status = queryObject.status;
364 |     id = queryObject.id;
365 |     console.log(
366 |         "\nBuyer/Seller requested to change the status of contract to " + status
367 |     );
368 |
369 |     result = await approveOrCancelAgreement(id, status);
370 |
371 | }
```

```
JS index.js  X

JS index.js > [ø] requestListener
365  |   console.log(
366  |     "\nBuyer/Seller requested to change the status of contract to " + status
367  |   );
368
369  |   result = await approveOrCancelAgreement(id, status);
370
371  |   console.log("Status changed successfully!");
372  |   res.writeHead(200);
373  |   res.end(result);
374 } else if (req.url.startsWith("/submit-agreement")) {
375   value = queryObject.value;
376   id = queryObject.id;
377   result = await createAsset(id, value);
378   res.writeHead(200);
379   res.end(result);
380 } else {
381   res.writeHead(200);
382   res.end("Unsupported URL");
383 }
384 ];
385
386 const server = http.createServer(requestListener);
387 server.listen(port, host, async () => {
388   await initNetwork008();
389   console.log(`Server is running on <a href="http://${host}:${port}`>);
390 });
391
```

Output

Figure 8 Output of DAPP

Figure 9 Output of DAPP cont..

- [2] Gorgin, “Digital signature in Node JS,” *Hashnode*, 08-Aug-2021. [Online]. Available: <https://hashnode.com/post/digital-signature-in-node-js-cks3ewmq40vqrhqs1axb9h4ci>. [Accessed: 08-Jul-2022].