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

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.

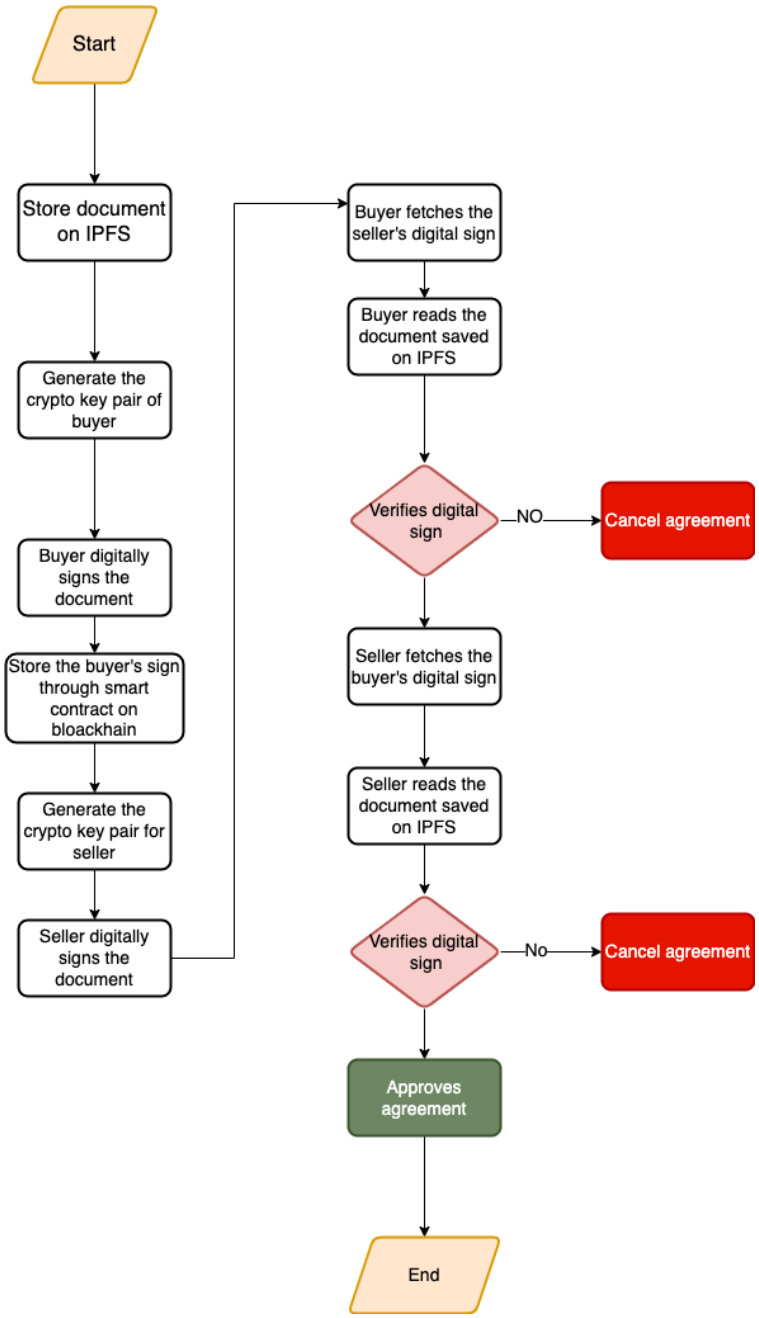


Figure 1 flow diagram

Figure 2 Hyperledger smart contract

notary-smart-contract008.js — smart-contract

notary-smart-contract008.js

lib > 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    /**
```

Figure 3 Hyperledger smart contract cont..

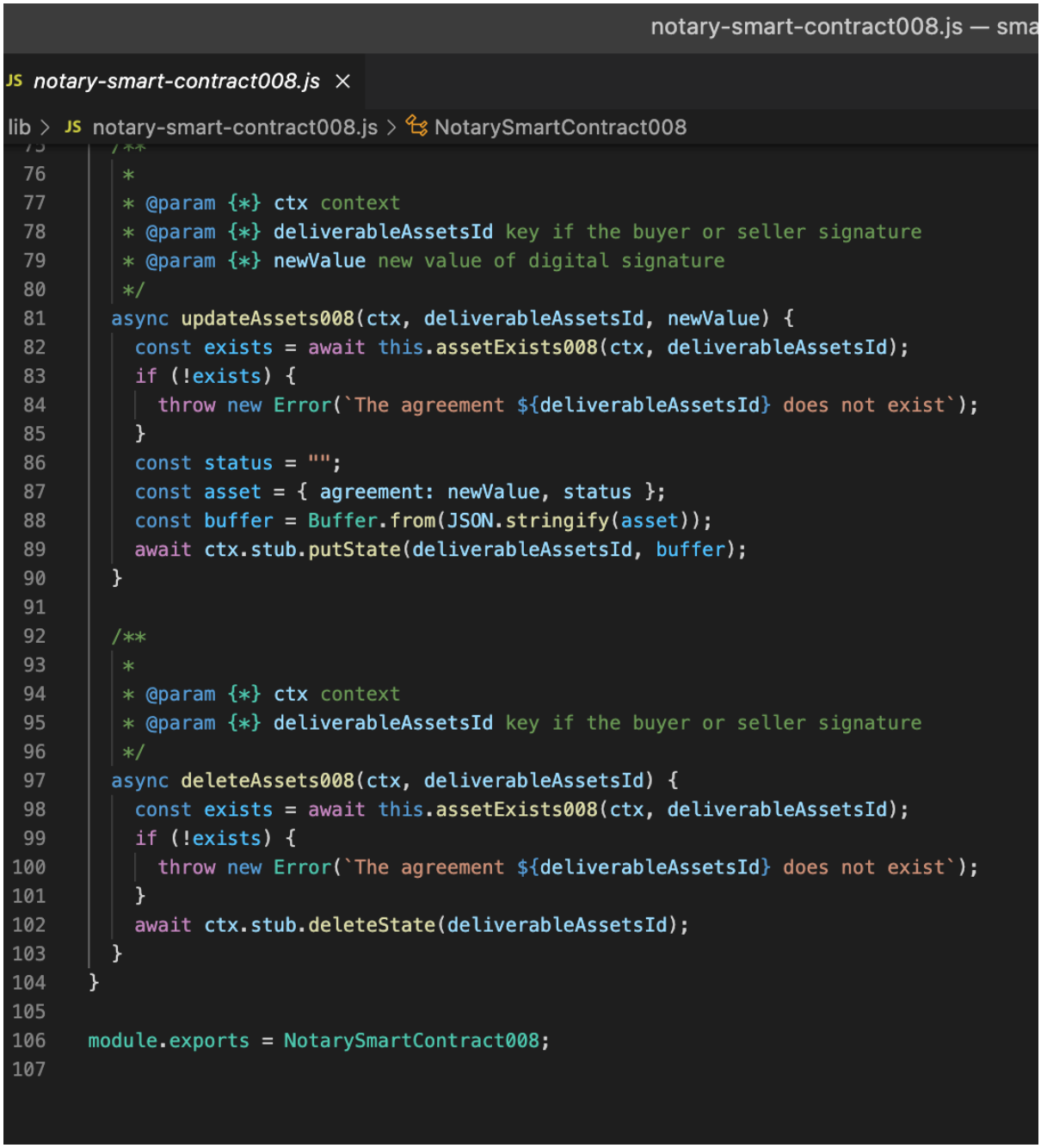


Figure 4 Hyperledger smart contract cont..

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.

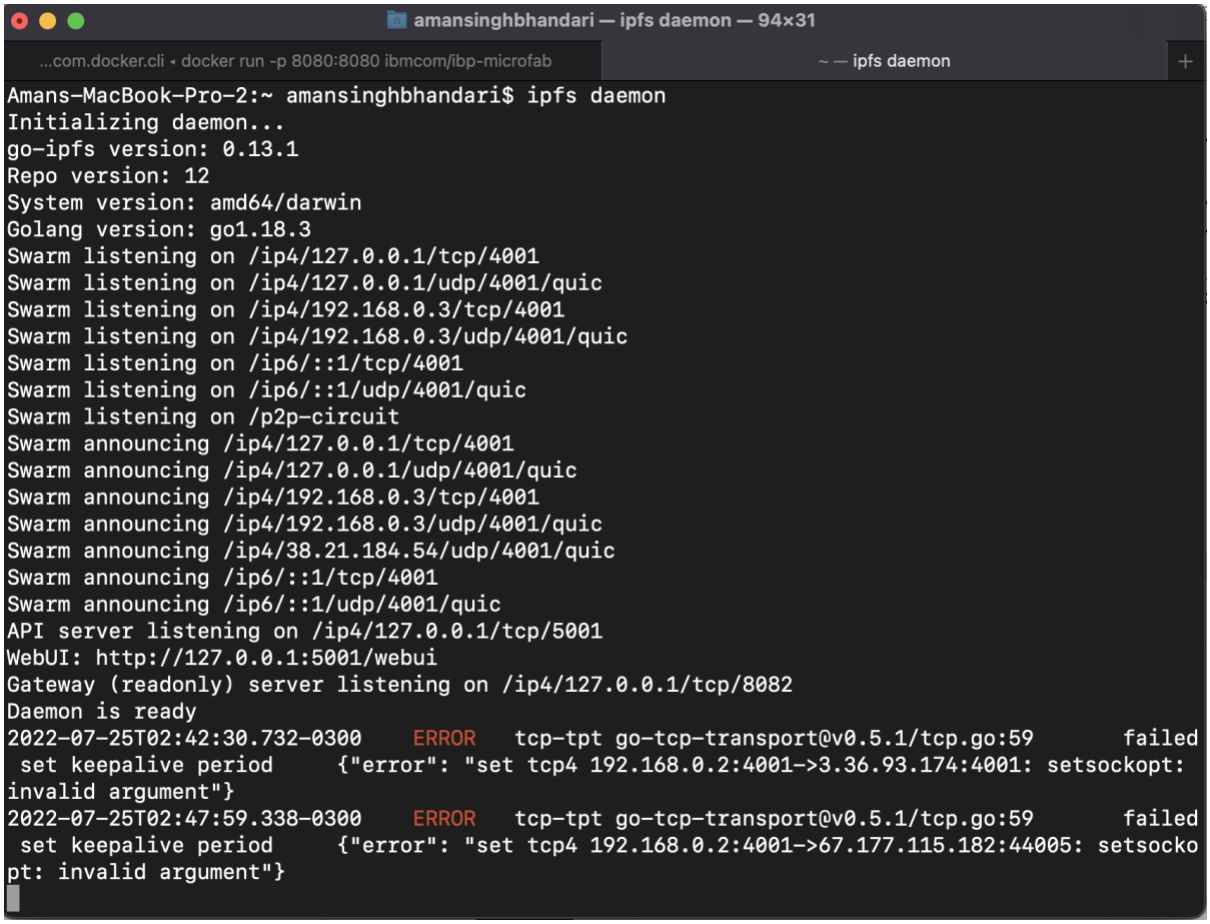


Figure 5 local IPFS

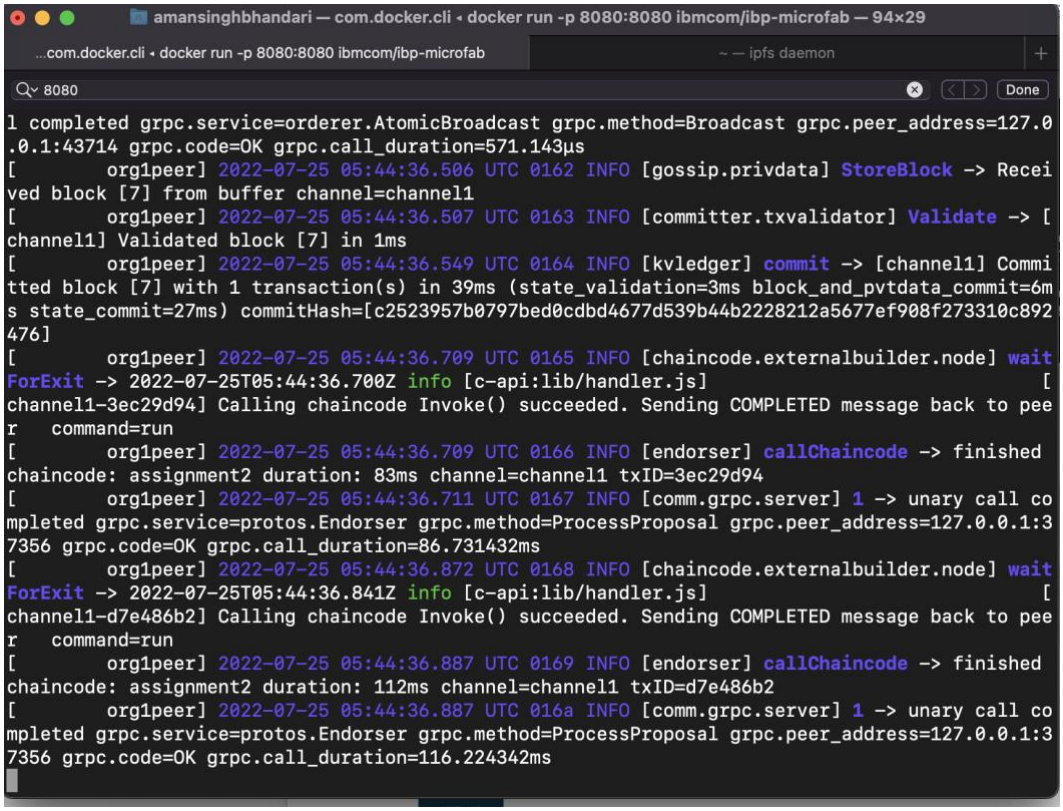


Figure 6 ibm-microfab docker container

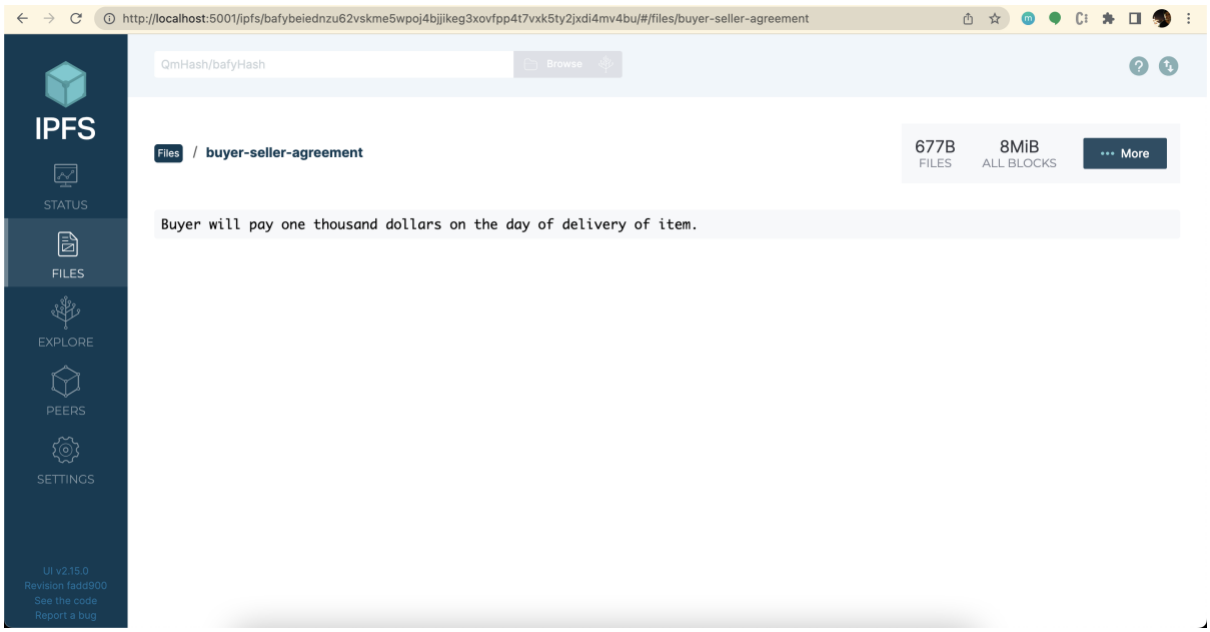


Figure 7 IPFS webui

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.

index.js

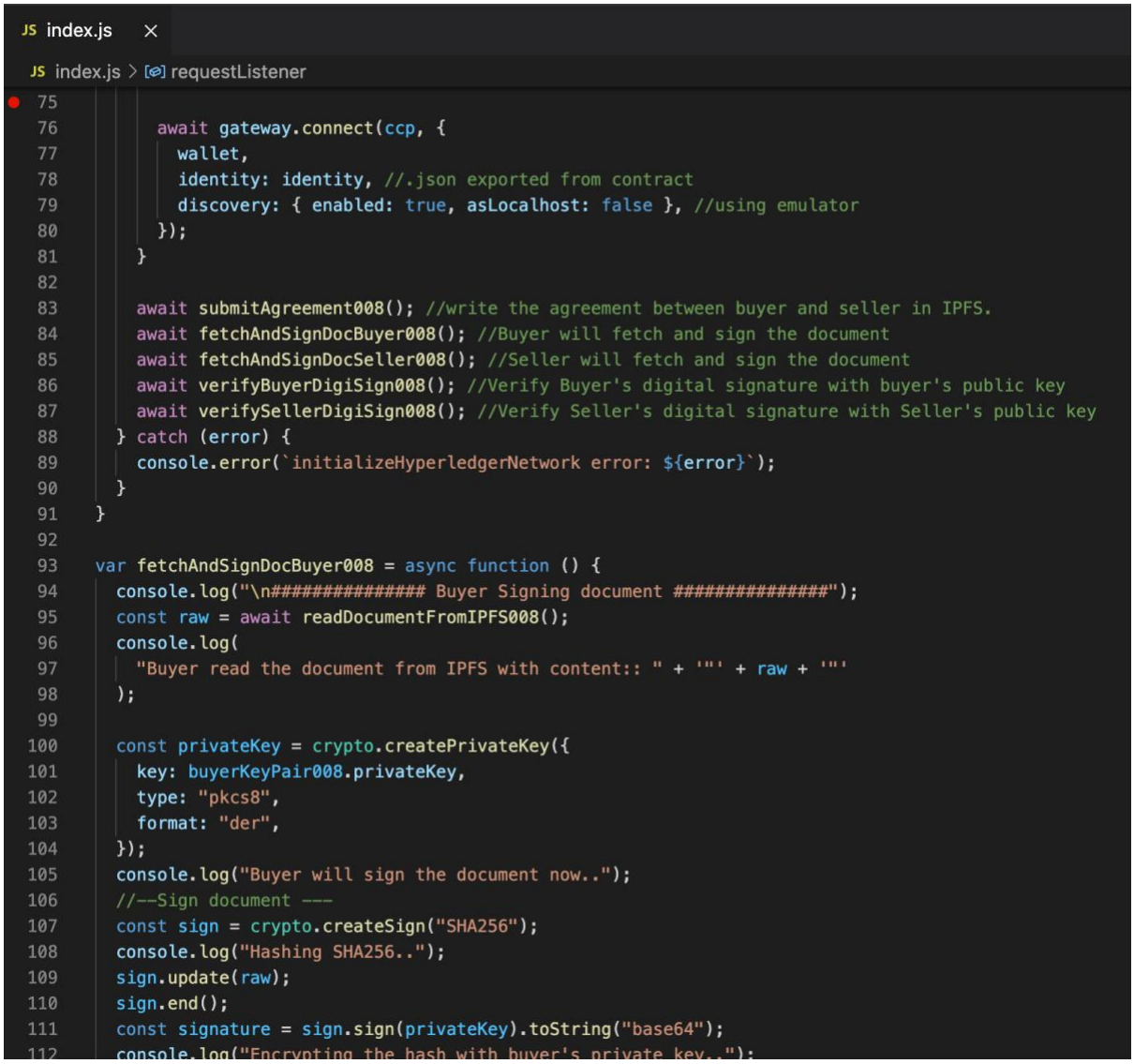
index.js > requestListener

```
/**
 * @author aman singh bhandari
 * @bannerid B00910008
 */
import { createRequire } from "module";
import { create } from "ipfs-http-client";
import { AbortController } from "node-abort-controller";
import { fileURLToPath } from "url";

const require = createRequire(import.meta.url);

var Web3 = require("web3");
const crypto = require("crypto");
const client = create();
var MFS_path008 = "/buyer-seller-agreement";
global.AbortController = AbortController;
let details = "{price:12ETH, item:rent-bmw3series}"; //Agreement between parties

const { Gateway, Wallets } = require("fabric-network");
const path = require("path");

const AppUtil = require("./lib/AppUtil.cjs");

const { buildWallet } = require("./lib/AppUtil.cjs");

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
const walletPath = path.join(__dirname, "Org1"); //Org1 is exported and is kept in root folder

const http = require("http"); //protocol
const url = require("url");

const host = "0.0.0.0"; //localhost
const port = 8083; //port to listen

let identity = "Org1 Admin";
let networkConnections = {};
let gateway = null;
```

index.js

index.js > requestListener

```
let gateway = null;
let network = null;
let contract = null;

const buyerKeyPair008 = crypto.generateKeyPairSync("rsa", {
  modulusLength: 2048,
  publicKeyEncoding: {
    type: "spki",
    format: "der",
  },
  privateKeyEncoding: {
    type: "pkcs8",
    format: "der",
  },
});

const sellerKeyPair008 = crypto.generateKeyPairSync("rsa", {
  modulusLength: 2048,
  publicKeyEncoding: {
    type: "spki",
    format: "der",
  },
  privateKeyEncoding: {
    type: "pkcs8",
    format: "der",
  },
});

async function initNetwork008() {
  try {
    const ccp = AppUtil.buildJunglekidsOrg1();
    const wallet = await buildWallet(Wallets, walletPath);
    if (gateway == null) gateway = new Gateway();

    if (network == null) {
      console.log("\nNetwork is not created yet.");
      console.log("Build a network gateway to connect to local emulator");

```


index.js
index.js > requestListener
75
76 await gateway.connect(ccp, {
77 wallet,
78 identity: identity, //.json exported from contract
79 discovery: { enabled: true, asLocalhost: false }, //using emulator
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..");

index.js

index.js > requestListener

```
    console.log("Encrypting the hash with buyer's private key..");
    console.log("Signature of buyer completed successfully!");
    console.log("Digital signature is ", signature);

    await createAsset("BuyerSignature", signature);
  };

  var fetchAndSignDocSeller008 = async function () {
    console.log("\n############### Seller Signing document ###############");
    const raw = await readDocumentFromIPFS008();
    console.log(
      "Seller read the document from IPFS with content:: " + '"' + raw + '"'
    );

    const privateKey = crypto.createPrivateKey({
      key: sellerKeyPair008.privateKey,
      type: "pkcs8",
      format: "der",
    });
    console.log("Seller will sign the document now..");
    //--Sign document ---
    const sign = crypto.createSign("SHA256");
    console.log("Hashing SHA256..");
    sign.update(raw);
    sign.end();
    const signature = sign.sign(privateKey).toString("base64");
    console.log("Encrypting the hash with seller's private key..");
    console.log("Signature of seller completed successfully!");
    console.log("Digital signature is ", signature);

    await createAsset("SellerSignature", signature);
  };

  var verifyBuyerDigiSign008 = async function () {
    console.log("\n############### Verify buyer's signatures ###############");

    const raw = await readDocumentFromIPFS008();
    console.log("Read plain text of document from IPFS..");
```

index.js

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

index.js

index.js > requestListener

```
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), {
```

index.js

index.js > requestListener

```
      .write(MFS_path008, new TextEncoder().encode(agreement008), {
        create: true,
      })
      .then(async (r) => {
        client.files.stat(MFS_path008, { hash: true }).then(async (r) => {
          let ipfsAddr008 = r.cid.toString();
          console.log("Stored file on IPFS with address ", ipfsAddr008);
        });
      })
      .catch((e) => {
        console.log(e);
      });
  };

async function initContract() {
  try {
    console.log(
      "\nFrom the gateway created to access the emulator, retreive the channel"
    );
    network = await gateway.getNetwork("channel1"); //get the channel
    console.log("Got the network gateway of specified channel");
    contract = network.getContract("assignment2"); //get contract to access its method later
    console.log("Got the contract");
    networkConnections["assignment2"] = contract; //add to connection map
    return contract;
  } catch (error) {
    console.error(`\nContract initialization error: ${error}`);
  }
}

/**
 *
 * verifies if the contract is loaded and
 */
async function getActorConnection() {
  if (!networkConnections["assignmentz"]) {
    await initContract();
  }
```

index.js

index.js > 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 a
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();
```

index.js

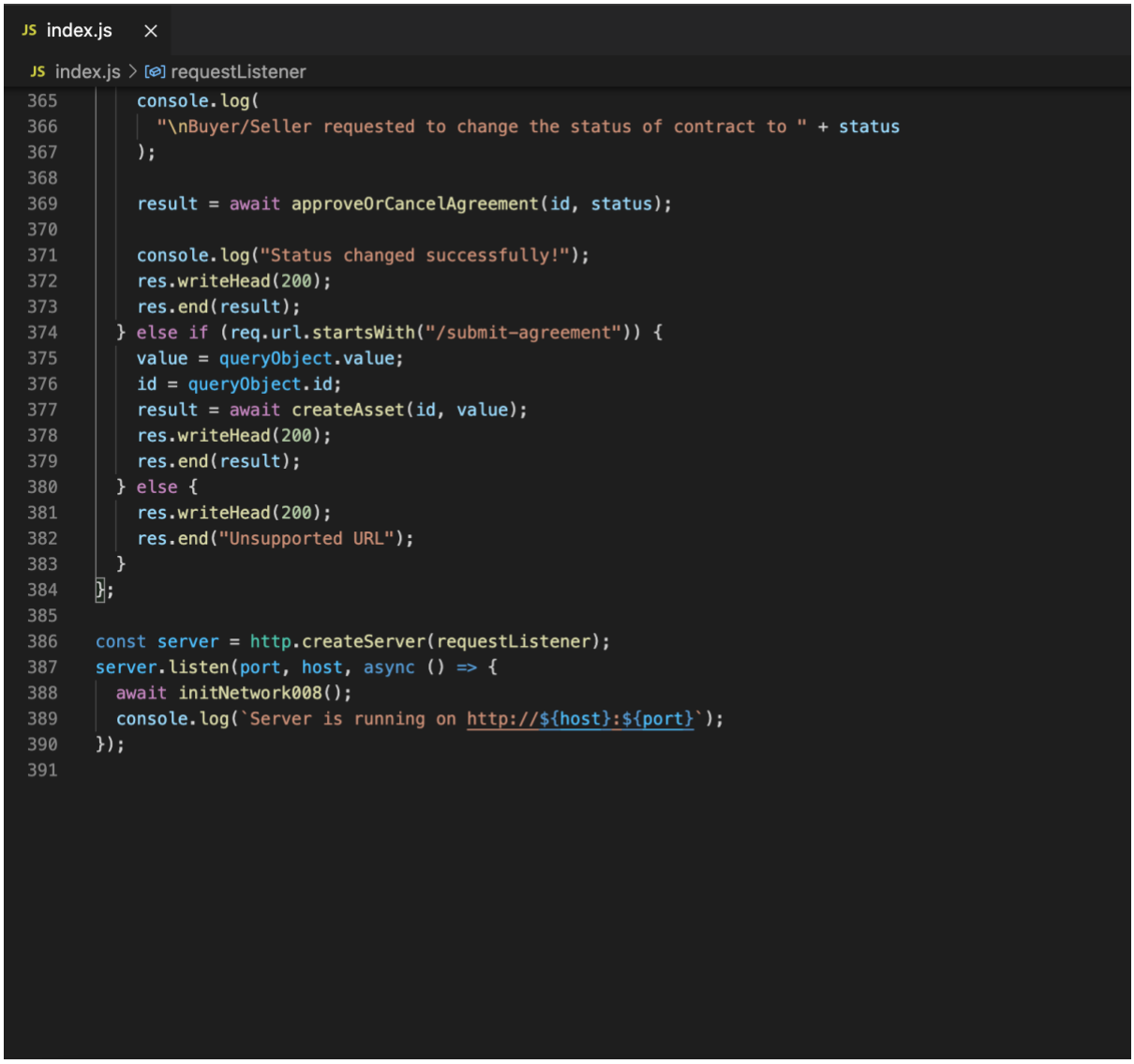
index.js > requestListener

```
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 retreving the Buyer/Seller got the below agreement and supporting fields \n" +
307         result
308     );
309     return result;
310   }
311
312   const 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 retrivedHash = resultjson.hash;
333       console.log("\nretrived agreement from the contract: " + agreement);
334       console.log("retrived hash from the contract: " + retrivedHash);
```

index.js

index.js > requestListener

```
    console.log("retrived hash from the contract: " + retrivedHash);
    console.log(
      "\nBuyer/Seller will now calculate hash of retreived agreement (md5): "
    );

    var calculatedHash = crypto
      .createHash("md5")
      .update(agreement)
      .digest("hex"); //Hashing with md5

    console.log("Calculated hash is: " + calculatedHash);

    if (calculatedHash === retrivedHash) {
      console.log(
        "\nYayy !!! Hash matched that means the agreement integrity is verified!"
      );
      console.log("Buyer/Seller will now APPROVE the agreement");
      resultjson.message =
        "Buyer retrived the asset -> calculated its hash -> It matched :) !!!";
    } else {
      console.log("\nOopsss !!! Hash didn't match");
      console.log("Buyer/Seller will now CANCEL the agreement");
      resultjson.message =
        "Buyer retrived the asset -> calculated its hash -> And it didn't match :( !!!";
    }
    result = JSON.stringify(resultjson);
    res.writeHead(200);
    res.end(result);
  } else if (req.url.startsWith("/approve-agreement")) {
    status = queryObject.status;
    id = queryObject.id;
    console.log(
      "\nBuyer/Seller requested to change the status of contract to " + status
    );

    result = await approveOrCancelAgreement(id, status);
```


index.js
index.js > 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 http://${host}:${port}`);
390 });
391

Output

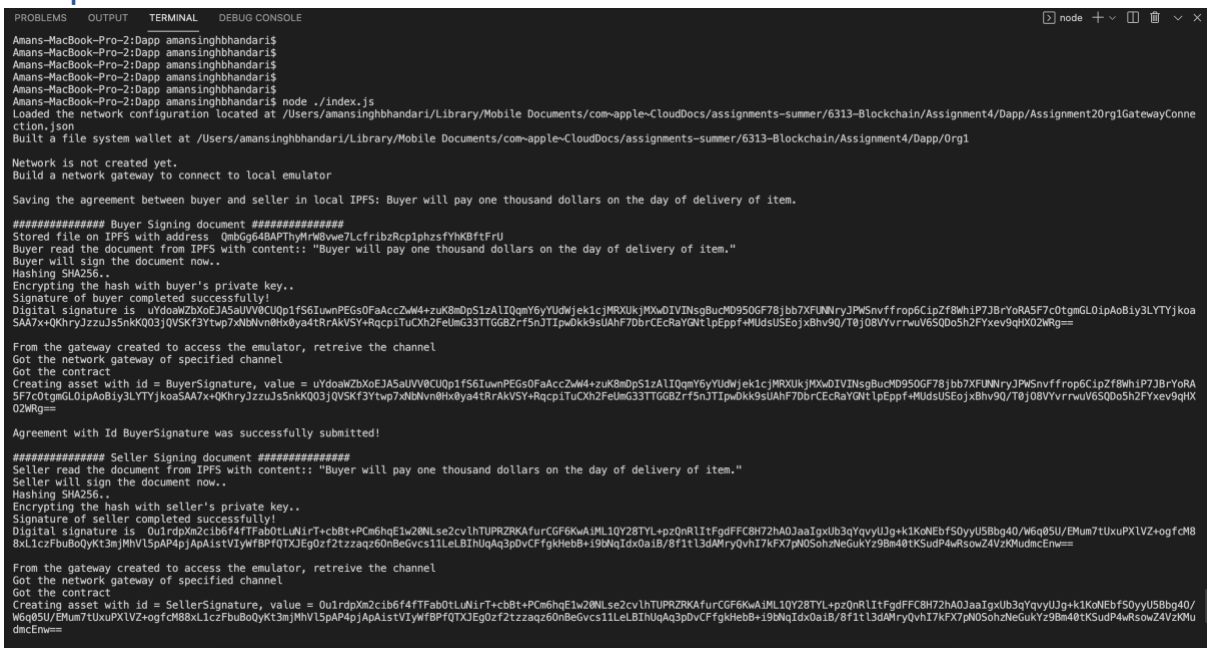


Figure 8 Output of DAPP

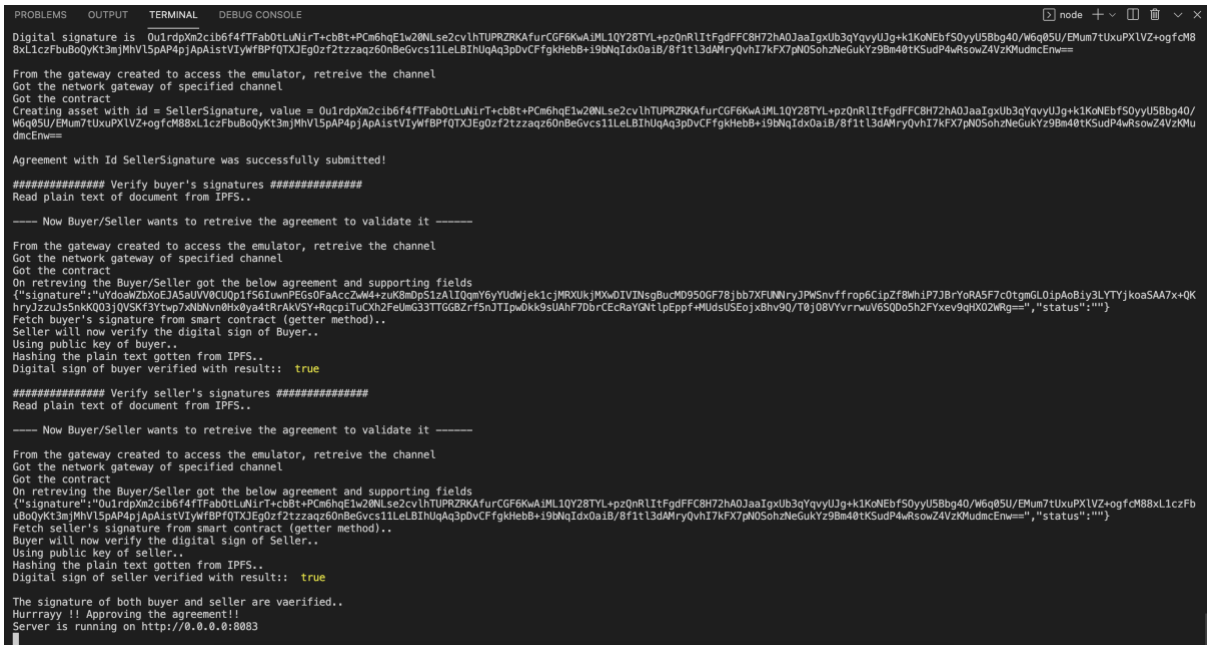


Figure 9 Output of DAPP cont..

References

[1] Chrisgangliu, "Chrisgangliu/6313-ipfs," *GitHub*. [Online]. Available: https://github.com/chrisgangliu/6313-ipfs. [Accessed: 08-Jul-2022].

[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].