POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica (Software Engineering)

Tesi di Laurea Magistrale

Crosschain with Hyperledger Fabric and Ethereum

Dapp build using Hyperledger Fabric and Ethereum networks

Relatore
prof.ssa Valentina Gatteschi

Candidato
Stefano Franzoni

Tutor aziendale
dott. Alfredo Favenza

Anno accademico 2019 – 2020

Indice

Elenco delle figure

Elenco delle tabelle

0.1 Oveview

0.1.1 Actors

The overview and flow of the Dapp developed is shown in **Figure 1**. The **Actors** involved in the system are:

- **User**: It's the *end user*. It use the web-app to send old clothes and purchase from Reclothes store
- **Reclothes Admin**: It's the *system admin*, it perform the actions in order to handle the system
- **Producer**: It's part of the upcycling process. It receive the materials to perform the recycling process

Each of that access to the system wich different permission and priviledges. Once the user is logged in, It could access to several features. It's possible to split the overview flow into 2 subflow starting from Reclothes actor, the **User side** and the **Producer side**.

1. **User Side**
   (a) User send Box with old clothes and receive Fabric points and ERC20 Token
   (b) User purchase items inside dapp store using Fabric points and ERC20 Token
2. **Producer Side**
   (a) Reclothes send clothes box with old matherials and receive Regeneration Credits
   (b) Reclothes spend the Regeneration Credits to purchase upcycled clothes by Producer

0.1.2 Token echanged

The **Token** exchanged over the networks will be:

1. **Over Fabric Net**
   (a) **User Token**: It's a point used to handle part of payment system, points based, related to clothes shipping from User to Reclothes and viceversa.
   (b) **Regeneration Credits**: It's a point used to handle part of credit system, points based, related to clothes shipping from Reclothes to Producers and viceversa.
2. **Over Ethereum Net**
   (a) **CO2 Token**: It's and ERC20 Token run over public network in charge to handle part of payments related to clothes shipping from User to Reclothes and viceversa.

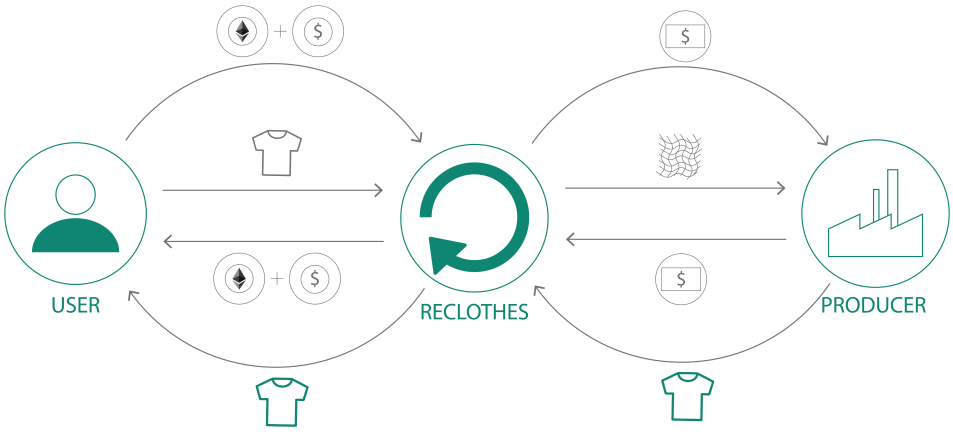

Figura 1. UseCase Overview

0.2 Technologies used to perform crosschain interaction

0.2.1 Tool Used

- **Metamask**: It's used as ethereum wallet to perform and sign the transactions started by dapp
- **Web3**: It's the software library used to interact with smart contract
- **Fab3 Proxy**: It map the Web3 API with the Fabric SDK in order to interact with Fabric network. It perform a mapping between the Fabric Identity (X.509) with an eth address, generated on the fly, used to perform dapp call.
- **Fabric Chaincode EVM**: It's the EVM chaincode that allow to run Solidity smart contract over the Fabric network
- **Expressjs**: Web Framework used to develop web-app and smart contract API
- **Infura**: allow to run a Ethereum node in order to set an endpoint used to interact with own contract.
- **Docker**: The fabric network components run inside Docker containers.

Figure **??** show The Architectural Flow and how the technologies is used and interact each other.

0.3 Use Cases

0.3.1 UseCase 1 - User Side

As shown in **Figure 2** both Actors User and Reclothes Admin, once is logged in, access to a set of features. The Use case diagram show all the action that both users could perform over the networks and tha flows that each actions follow. The features are split over the two network, the Fabric one and the Ethereum one.

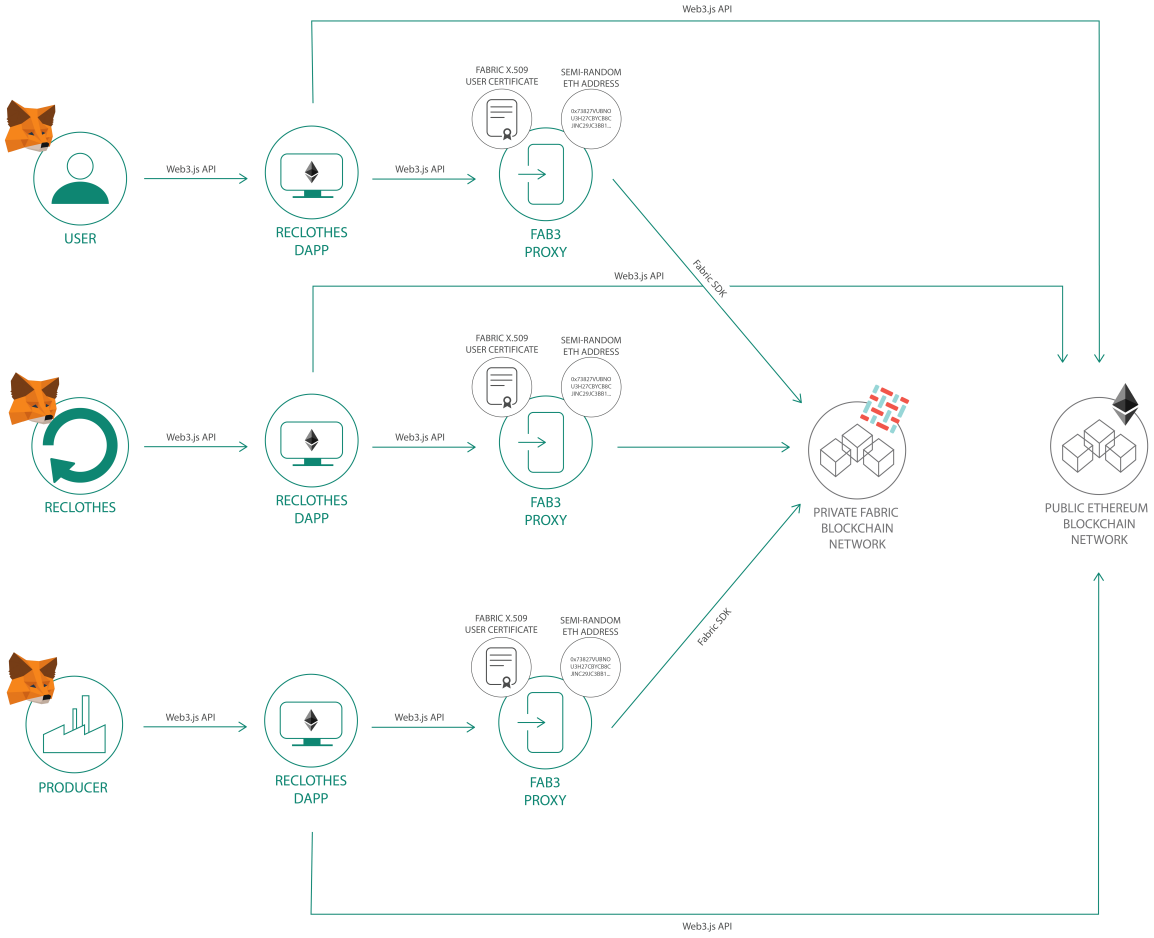

Figura 2. Architectural Flow

The Internal Flow of the **Send Box** macro process is the follow one:

1. User send box with old clothes
2. Reclothes Admin receive box, evaluate it
3. The web app perform the payments from Reclothes Account to User Account
4. Once both transactions succed, both token are accredited and User could spend it

Transactions

The two process that start the transactions are the **Evaluation** performed by The Reclothes Admin and the **Purchase Items** performed by the Users over the platform store.

1. The **Evaluation** process works in the following way:

   (a) Reclothes Admin visualize the next pending request to be evaluated
   (b) Reclothes Admin evaluate it and set an amount value of Fabric points and ERC20 Token to be send
      i. The points are sent over Fabric network invoking a chaincode function
      ii. The ERC20 Token are send over Ethereum network (Ropsten), sending the token to the ETH Address stored in the smart contract during the User Registration Phase.
   (c) Once both transactions succed, both tokens are accredited and User could spend it

2. The **Purchase Items** process in a similar way:
   (a) User choose the items to purchase over the web-app store
   (b) Reclothes Admin evaluate it and set a value amount of Fabric points and ERC20 Token to send
      i. The points are sent over Fabric network invoking a chaincode function
      ii. The ERC20 Token are send over Ethereum network (Ropsten), sending the token to the ETH Address stored in the smart contract during the User Registration Phase.
   (c) Once both transactions succed, both token are accredited and User could spend it

0.3.2 UseCase 2 - Producer Side

The Use case diagram shown in **Figure 3** describe how Reclothes Admin and Producer interact each other. In this case all the features are performed over the Hyperledger Fabric network.

We could split the flow into two subflow, the first one from Reclothes to Producer, that we could identify with the two actions *Send Box* and *Purchase Box*, on the other hand the second subflow from Producer to Reclothes is summarize into *Evaluate Material* function.

All the asset exchange is handle using **Regeneration Credits** a Fabric token exchanged and handle by the Fabric chaincode and running over Fabric network.

1. **from Reclothes to Producer**
   (a) **Send Box**
      i. Send Box with old materials to be recycled by the Producer Company
      ii. Receive **Regeneration Credits** based on the old materials evaluation
   (b) **Purchase Box**
      i. Reclothes Admin purchase Box by Producer Company relized with recycled materials, spending the Regeneration Credits
2. **from Producer to Reclothes**
   (a) **Evaluate Material**
      i. Producer Admin perform the evaluation of the materials received by Reclothes, after the evaluation the transactions of Regeneration Credits is performed by the Fabric chaincode

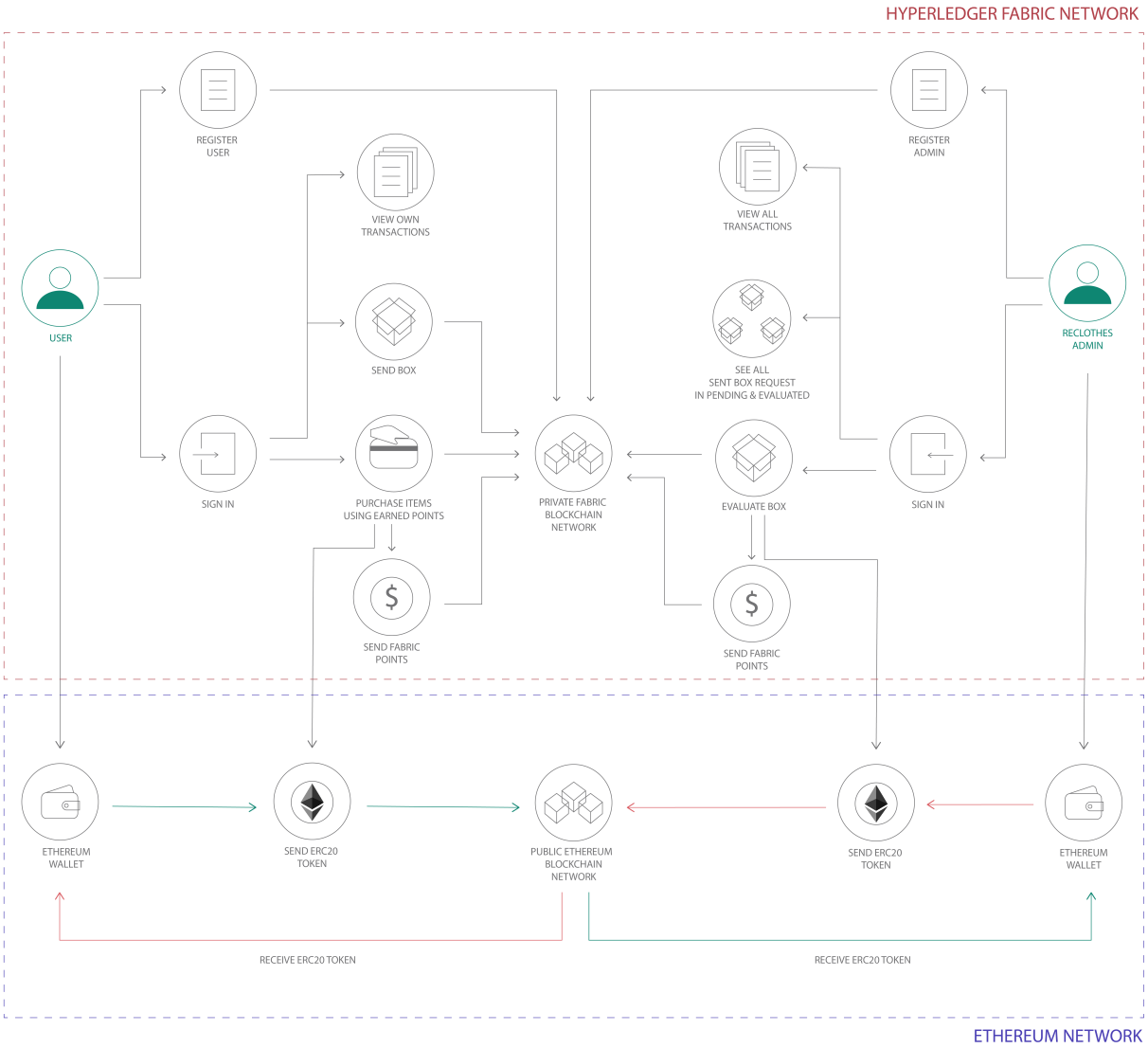

Figura 3. UseCase 1

0.4 Smart Contract

All the smart contract are developed in Solidity.

Both the Fabric contracts are use the `fabric-chaincode-evm` in order to allow Solidity code into fabric network

1. **Hyperledger Fabric**
   (a) **User Contract**: handle the the User side, registration and interactions phase.
   (b) **Producer Contract**: handle the interaction from Reclothes to Producers.

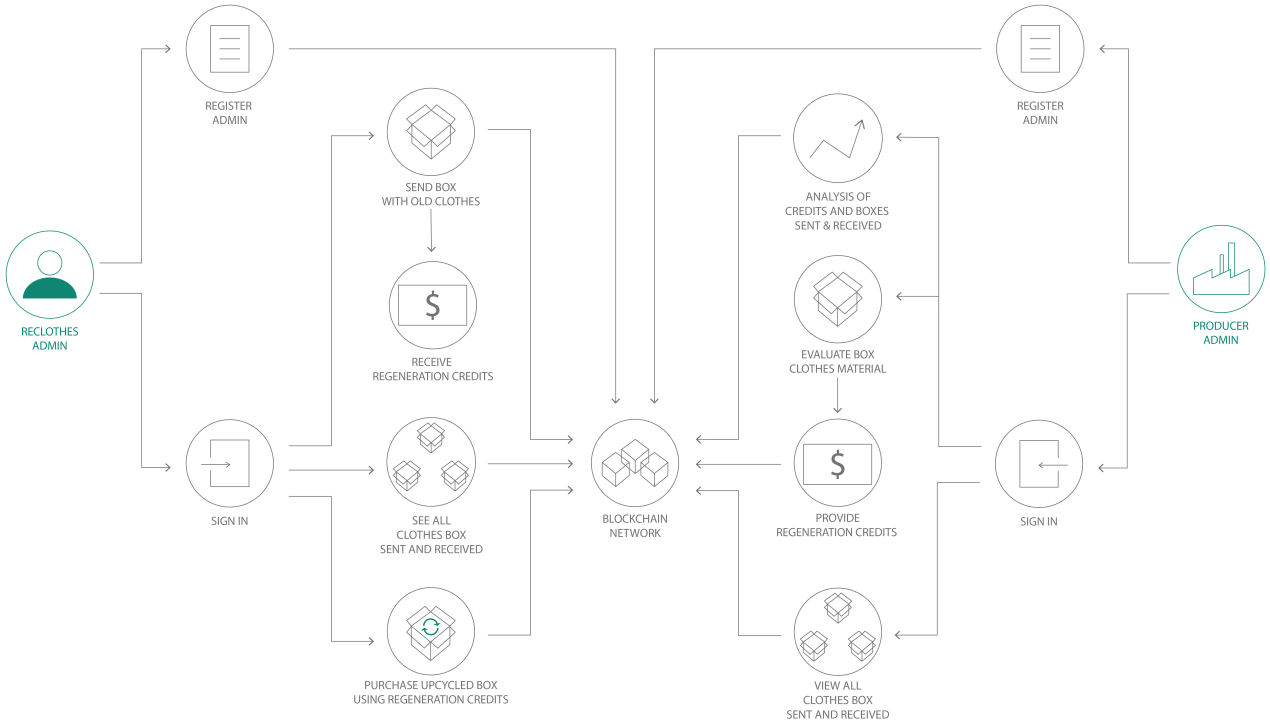

Figura 4. UseCase 2

2. **Ethereum**

   (a) **ERC20 Contract**: it's a standard smart contract with a Max Supply fixed to 100.000.000 . The Contract run over Ropsten network and we access to it through the Infura node.

0.4.1 User Contract

Data Structure The model of the data structures is divided inside 4 structs:

1. **User**: model all users data
2. **Admin**: model Reclothes Admin data
3. **PointsTransaction**: Model transactions data and incorporate `TransactionType` used to identify the flows
4. **ClothesBox**: The box sent with old clothes

```
1         // model a user
2         struct User {
3             address userAddress;    // User address (inside fabric environment)
4             address publicAddress;  // external eth public address of User Admin
5             string firstName;
6             string lastName;
```

```
7             string email;
8             uint points;             // Fabric points amount
9             bool isRegistered;       // Flag for internal use
10            uint numTransaction;     // number of transactions performed
11            mapping(uint => PointsTransaction) userTransactions;
12            uint numBox;             // number of box transaction evaluated
13            mapping(uint => ClothesBox) box;
14        }
15
16        // model a admin
17        struct Admin {
18            address adminAddress;    // Admin address (inside fabric environment)
19            address publicAddress;   // external eth public address of Admin
20            string name;
21            bool isRegistered;       // Flag for internal use
22        }
23
24        // model points transaction
25        enum TransactionType { Earned, Redeemed }
26        struct PointsTransaction {
27            uint points;
28            TransactionType transactionType;
29            address userAddress;     // user address involved
30            address adminAddress;    // admin address involved
31        }
32
33        // model clothes box to ship
34        struct ClothesBox {
35            address userAddress; // reclothes-producer Admin
36            uint tshirt;         // Number of item
37            uint pants;          // Number of item
38            uint jacket;         // Number of item
39            uint other;          // Number of item
40            bool isEvaluated;    // Flag to check if box evaluation is performed
41            uint points;         // fabric value amount of the box
42        }
```

Transactions There's two functions that performs transaction from and to Reclothes:

1. **earnPoints**: It's an internal function called by `sendBox` function. It performs the fabric points transaction from Reclothes to User.
2. **usePoints**: performs the fabric points transaction when the User purchase item by Reclothes store.

```
1     //update users with points earned
2     function earnPoints (uint _points, address _userAddress ) onlyAdmin(msg.sender
          ) internal {
3
4       // verify user address
5       require(users[_userAddress].isRegistered, "User address not found");
6
7       // update user account
8       users[_userAddress].points = users[_userAddress].points + _points;
9
10      PointsTransaction memory earnTx = PointsTransaction({
11        points: _points,
12        transactionType: TransactionType.Earned,
13        userAddress: _userAddress,
14        adminAddress: admins[msg.sender].adminAddress
```

```
15        });
16
17        // add transction
18        transactionsInfo.push(earnTx);
19
20        users[_userAddress].userTransactions[users[_userAddress].numTransaction] =
              earnTx;
21        users[_userAddress].numTransaction++;
22
23        usersTransactions[totTx] = earnTx;
24        totTx++;
25
26      }
27
28
29      //Update users with points used
30      function usePoints (uint _points) onlyUser(msg.sender) public {
31
32        // verify enough points for user
33        require(users[msg.sender].points >= _points, "Insufficient points");
34
35        // update user account
36        users[msg.sender].points = users[msg.sender].points - _points;
37
38        PointsTransaction memory spendTx = PointsTransaction({
39          points: _points,
40          transactionType: TransactionType.Redeemed,
41          userAddress: users[msg.sender].userAddress,
42          adminAddress: 0
43        });
44
45        // add transction
46        transactionsInfo.push(spendTx);
47
48        users[msg.sender].userTransactions[users[msg.sender].numTransaction] =
              spendTx;
49        users[msg.sender].numTransaction++;
50
51        usersTransactions[totTx] = spendTx;
52        totTx++;
53      }
```

0.4.2 Producer Contract

Data Structure The model of the data structures is divided inside 3 structs:

1. **Producer**: model all producers data
2. **Admin**: model Reclothes Admin data
3. **ClothesBox**: The box sent with old clothes

```
1       // model a producer
2       struct Producer {
3           address adminAddress;    // Producer Admin address (inside fabric
                environment)
4           address publicAddress;  // external eth public address of Producer Admin
5           string name;            // Producer admin name
6           bool isRegistered;      // Flag for internal use
7           uint numBox;            // number of box transactions evaluated
```

```
        uint pointsProvided;    // amount of points provided by own evaluations
        mapping(uint => ClothesBox) box;
    }

    // model a admin
    struct Admin {
        address adminAddress;   // Admin address (inside fabric environment)
        address publicAddress;  // external eth public address of Admin
        string name;            // Admin name
        bool isRegistered;      // Flag for internal use
        uint numBox;            // number of box transaction evaluated
        uint creditSpent;       // amount of points provided by own evaluations
        mapping(uint => ClothesBox) box;
    }

    struct ClothesBox {
        address adminAddress; // reclothes-producer Admin
        uint tshirt;         // Number of item
        uint pants;          // Number of item
        uint jacket;         // Number of item
        uint other;          // Number of item
        bool isEvaluated;    // Flag to check if box evaluation is performed
        uint points;         // fabric value amount of the box

        //mapping(uint => Clothes) clothes;
    }
```

Transactions There's two functions that perform transactions from and to Reclothes:

1. **evaluateBox**: function called by Producer to evaluate box materials, sent by Reclothes, to send Regeneration Credits.
2. **buyUpcycledBox**: function called by Reclothes Admin that spend earned Regeneration Credits to purchase upcycled clothes by Producer.

```
    // Evaluate Old Box
    function evaluateBox(uint _points) onlyProducer() public {
        //check correct pending request index
        require(evaluatedIndex < pendingIndex, "No more pending request");

        //check if evaluation is done
        require(!pendingBox[evaluatedIndex].isEvaluated, "Request just evaluated")
            ;

        //pop pending request
        ClothesBox storage box = pendingBox[evaluatedIndex];

        //update box transaction
        box.isEvaluated = true;
        box.points = _points;

        //add evaluated box
        evaluatedBox[evaluatedIndex] = box;
        evaluatedIndex++;

        debtPoints += _points;
        totPointsProvided += _points;
    }

```

```
24      function buyUpcycledBox(uint _tshirt, uint _pants, uint _jackets, uint _other,
            uint _points) onlyAdmin() public {
25          require(debtPoints >= _points, "Not enought credits accumulated in old
                material boxes");
26
27          ClothesBox memory box = ClothesBox({
28           adminAddress: msg.sender,
29           tshirt: _tshirt,
30           pants: _pants,
31           jacket: _jackets,
32           other: _other,
33           isEvaluated: true,
34           points: _points
35          });
36
37          admins[msg.sender].box[admins[msg.sender].numBox] = box;
38          admins[msg.sender].numBox++;
39          admins[msg.sender].creditSpent += _points;
40
41          //add upcycled box
42          upCycledBox[upCycledIndex] = box;
43          upCycledIndex++;
44
45          debtPoints -= _points;
46          totBoxNew++;
47      }
```

0.4.3 ERC20 Contract

It's a standard ERC20 token with the following features:

- **Symbol**: CO2
- **Name**: CarbonToken
- **Total supply**: 100000000
- **Decimals**: 18

The main features of the contract are describer by ERC20 interface

```
1       contract ERC20Interface {
2           function totalSupply() public constant returns (uint);
3           function balanceOf(address tokenOwner) public constant returns (uint
                balance);
4           function allowance(address tokenOwner, address spender) public constant
                returns (uint remaining);
5           function transfer(address to, uint tokens) public returns (bool success);
6           function approve(address spender, uint tokens) public returns (bool
                success);
7           function transferFrom(address from, address to, uint tokens) public
                returns (bool success);
8
9           event Transfer(address indexed from, address indexed to, uint tokens);
10          event Approval(address indexed tokenOwner, address indexed spender, uint
                tokens);
11      }
```

0.5 Network Architecture

The network Architecture build for the application includes:

1. **1 Orderer** organization with *1 ordered* running
2. **3 Organizations** each with 1 peer, Peer0, running
   (a) *Org1*: User Organization
   (b) *Org2*: Reclothes Admin Organization
   (c) *Org3*: Producer Organization
3. **2 Channels**
   (a) *Chanel12*: It's the cannel between Org1 and Org2 and allow the comunication between User and Reclothes
   (b) *Chanel23*: It's the cannel between Org2 and Org3 and allow the comunication between Reclothes and Producer

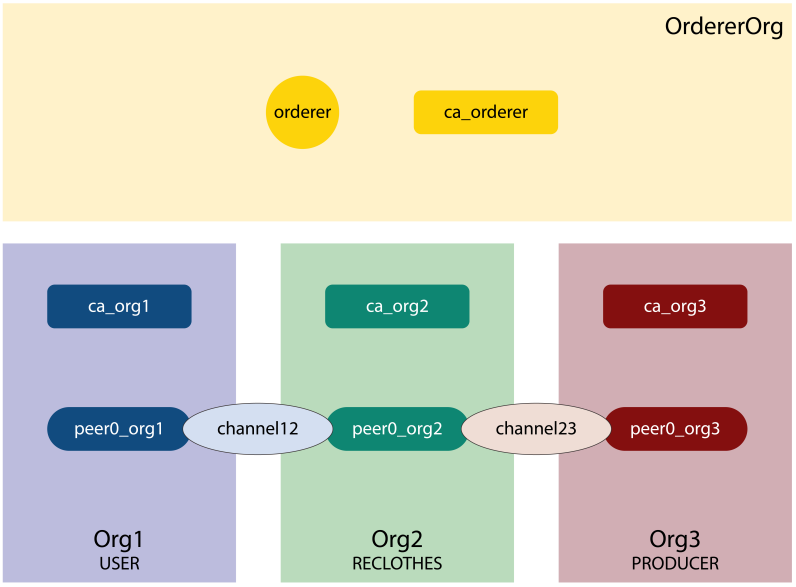

Figura 5. Fabric Network

0.5.1 Fabric Network

The Figure show how components interact each other. We could separate components into 2 categories, inside and outside Fabric Network. First of all we need to describe the components involved :

- **Web3 App**: It's the Dapp and the Client connection to the network

- **Channel**: It's the channel above which transfert data
- **CA**: It's the Certification Authority in charge of release certificates.
- **Peer**: It's "Fabric node", the endpoint of the internal network. It own by specific CA with fixed permissions, linked to the connected channels.
- **evm SC**: It's the Ethereum Virtual Machine Chaicode, used to run Solidity Smart Contract. The chaincode is installed over the peer.
- **ledger**: It's the ledger associated to the channel connected. There's a 1 to 1 association between ledger and channel.
- **CC**: It's the *Consortium*, It's associated to the channel, manage ownerships and It include a set of Organizations allowed.
- **Docker**: The network components run inside docker containers.

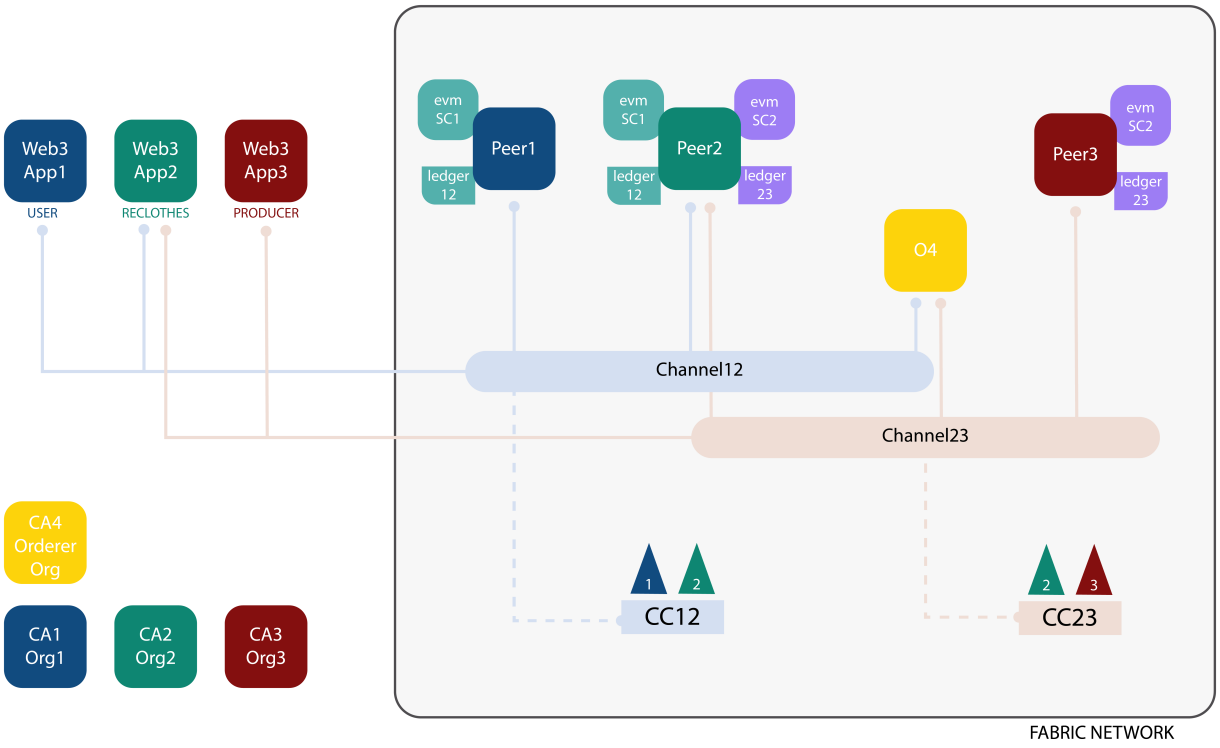

Figura 6. Fabric Network Components

The *chaincode* it's invoked calling the evm chaincode by the *App* Client, using the channel communication. Than the chaincode installed over the peer once is invoked agreed to the request and invoke the chaincode("smart contract") method. Once the method returns, the chaincode forward the reply to the App client. The Dapp forward the answer to the *Orderer* peer that validate the response, create a new block, add it to the chain, communicate it to the peer in order to syncronize the network and updating the Ledger World State.

Config File

To design and set up network components and rules, It's wrote the `config.yaml` file. The network is structured in the following lines of code:

```
Organizations:
- &OrdererOrg
    Name: OrdererOrg
    ID: OrdererMSP
    MSPDir: crypto-config/ordererOrganizations/example.com/msp
    Policies:
        Readers:
            Type: Signature
            Rule: "OR('OrdererMSP.member')"
        Writers:
            Type: Signature
            Rule: "OR('OrdererMSP.member')"
        Admins:
            Type: Signature
            Rule: "OR('OrdererMSP.admin')"

- &Org1
    Name: Org1MSP
    ID: Org1MSP
    MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
    Policies:
        Readers:
            Type: Signature
            Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
        Writers:
            Type: Signature
            Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
        Admins:
            Type: Signature
            Rule: "OR('Org1MSP.admin')"
    AnchorPeers:
        - Host: peer0.org1.example.com
          Port: 7051

- &Org2
    Name: Org2MSP
    ID: Org2MSP
    MSPDir: crypto-config/peerOrganizations/org2.example.com/msp
    Policies:
        Readers:
            Type: Signature
            Rule: "OR('Org2MSP.admin', 'Org2MSP.peer', 'Org2MSP.client')"
        Writers:
            Type: Signature
            Rule: "OR('Org2MSP.admin', 'Org2MSP.client')"
        Admins:
            Type: Signature
            Rule: "OR('Org2MSP.admin')"
    AnchorPeers:
        - Host: peer0.org2.example.com
          Port: 8051

- &Org3
    Name: Org3MSP
    ID: Org3MSP
    MSPDir: crypto-config/peerOrganizations/org3.example.com/msp
    Policies:
        Readers:
```

```
                Type: Signature
                Rule: "OR('Org3MSP.admin', 'Org3MSP.peer', 'Org3MSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org3MSP.admin', 'Org3MSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org3MSP.admin')"
        AnchorPeers:
            - Host: peer0.org3.example.com
              Port: 9051


                                        ...
                                        ...
                                        ...

Profiles:

    OrdererGenesis:
        <<: *ChannelDefaults
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org1
                    - *Org2
                    - *Org3

    Channel12:
        Consortium: SampleConsortium
        <<: *ChannelDefaults
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
            Capabilities:
                <<: *ApplicationCapabilities
    Channel23:
        Consortium: SampleConsortium
        <<: *ChannelDefaults
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org2
                - *Org3
            Capabilities:
                <<: *ApplicationCapabilities
```

End to End Interactions

Going deeper, the Figure show the flow of the end to end communication. How all the components are boxed inside the Peer component. The Fab3 map the web3 request and forward it to fabric peer. the request arrive to the evmcc that invoke Solidity smart contract methods.

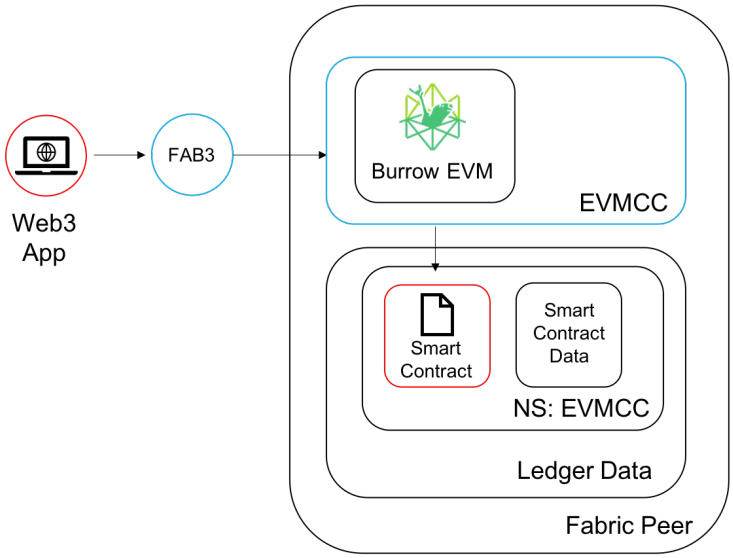

Figura 7. End To End

Chaincode Invocations

The Figure below describe the internal workflow of the chaincode invocation, where's involved the *Client* the *Peer* and the *Orderer*. All the information are transfer over the setted channel and in our study case, the client doesn't interact directly, but using *Fab3 Proxy* as intermediary.

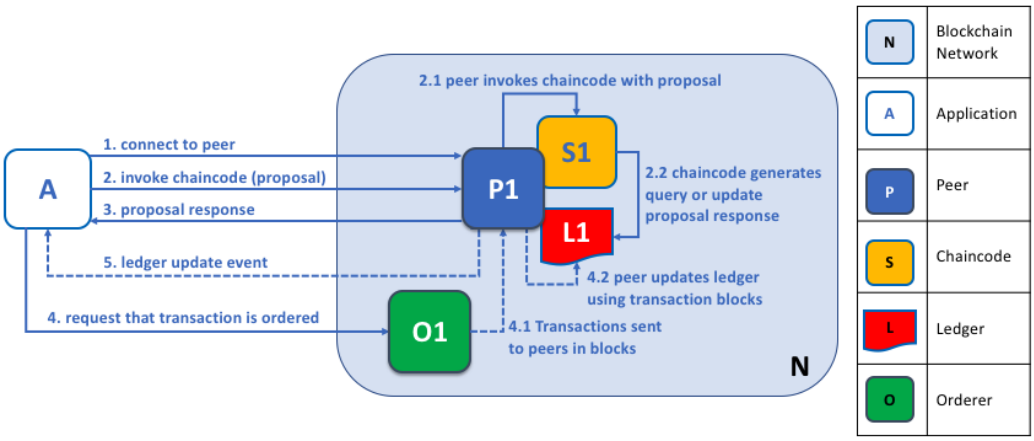

Figura 8. Smart Contract Invocation Process

0.5.2 Ethereum Network - Ropsten

To run the ERC20 token it's used the testnet Ropstan against the mainnet. To set up and upload own ERC20 Token over the ethereum network it's used:

- **My Ether Wallet**: To upload ERC20 contract
- **Etherscan.io**: To monitor and analyze transactions over the network
- **Metamask**: To create user wallets
- **Infura**: To set up a node in order to use it as endpoint and communicate with the Ropsten network, it is used as *Provider* in *Web3* library.

0.6 Dapp - Client

0.6.1 Thecnologies used

To develop the client application is used the following Technologies:

- **Expressjs**: It's a node.js framework that allow to develop api for own application
- **Bootstrap**: To build a user friendly front-end in order to interact in the best way
- **Web3**: Ethereum Javascript API, It's is a collection of libraries that allow you to interact with a local or remote ethereum node
  - **web3 0.20.2**: used for dapp developments, fabric side, It's a stable version and it's the version used in `fabric-chaincode-evm` development
  - **web3 1.0.0**: used for ethereum transactions, It's a version with more functionalities but less stable.

Starting from the Homepage the User is allowed to register itself as **User**, **Reclothes Admin** or **Producer**.

0.6.2 Core part of the web-app

The technical files and flow that dapp follow to run up it's the following one.

1. **Contract Address Generation**:
   (a) This step is in charge to run a script that deploy the contract adresses to be reffered during the app running.
      i. **UserContract.js**: running the script using node command, it return the address of the deployed contract
      ii. **ProducerContract.js**: running the script using node command, it return the address of the deployed contract
2. **dapp.js**: there's the core file that handle the contracts invokations, set up the contract address referance, and connect to a specific Fab3 instance.
3. **app.js**: It set up the API called by the web-app, map the request and forward to `dapp.js`.

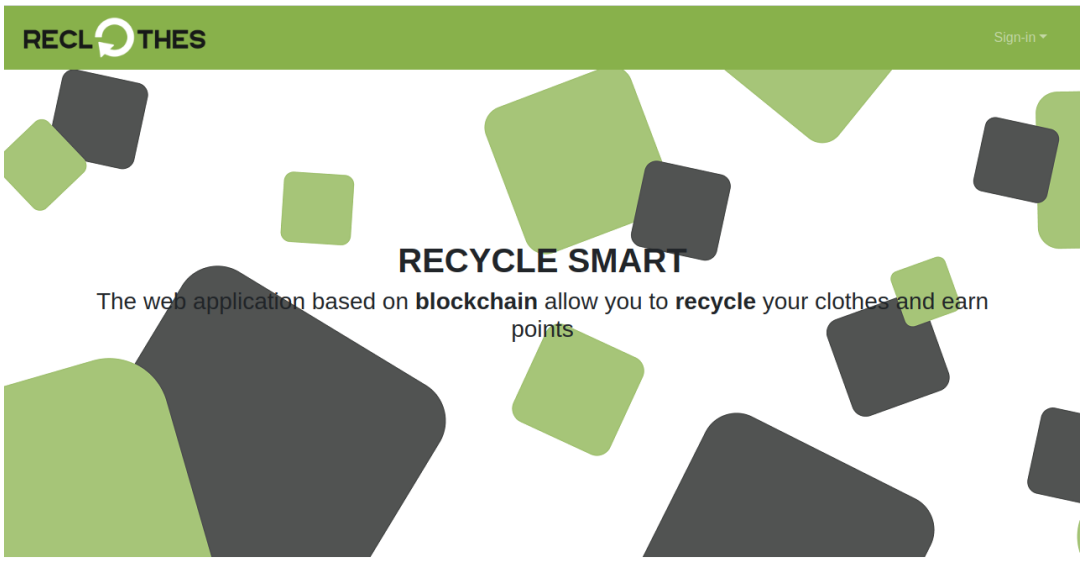

Figura 9. Home

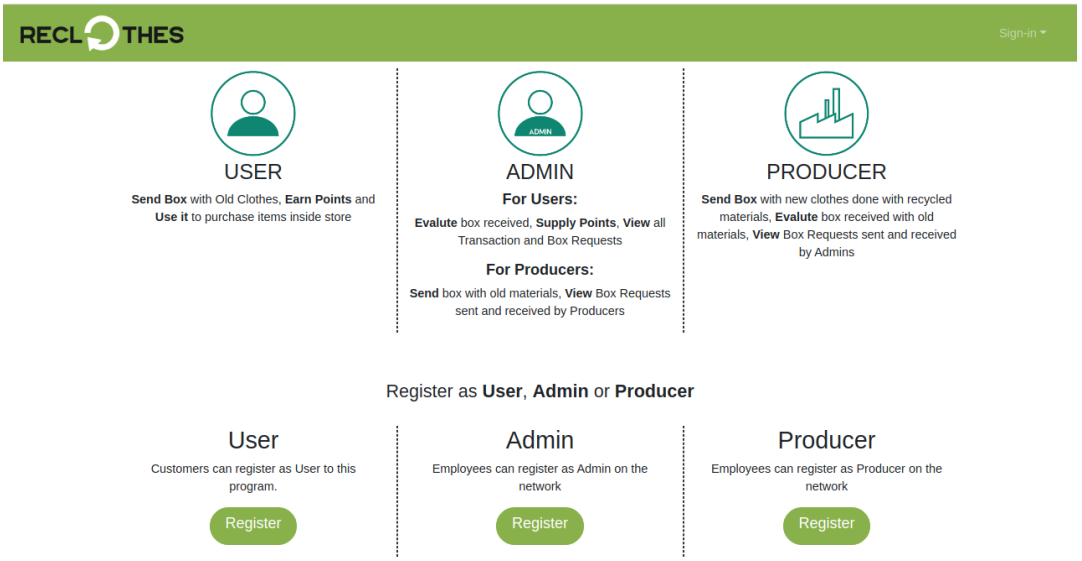

Figura 10. Registration Phase

0.6.3 Views

Homepage

The homepage allow user to view the feature of each User type and to access to the registration page.

User Page

The User page allow to view an overview infos once the user is logged in.

- **Address**: It's the public eth address setup during registration phase.

- **Points Balance**: It's the Fabric points balance earned by the user sending the boxes.

- **ERC20 Balance**: It's the eth balance of the public token running over eth network.

The Figure xxxx show how to compile the form in order to send box with old clothes. It's a simulation of the real process to sending box, that in the real case could be implemented thow a QRCode or RFID placed over the boxes.

The Figure xxx show how should be the store, purchase items over the platform start the transaction process.

There's other section about infos that user is allowed to see. **Transactions** performed over the fabric network and **Box Requests** that's all the history about the box sent and received with all the related informations.

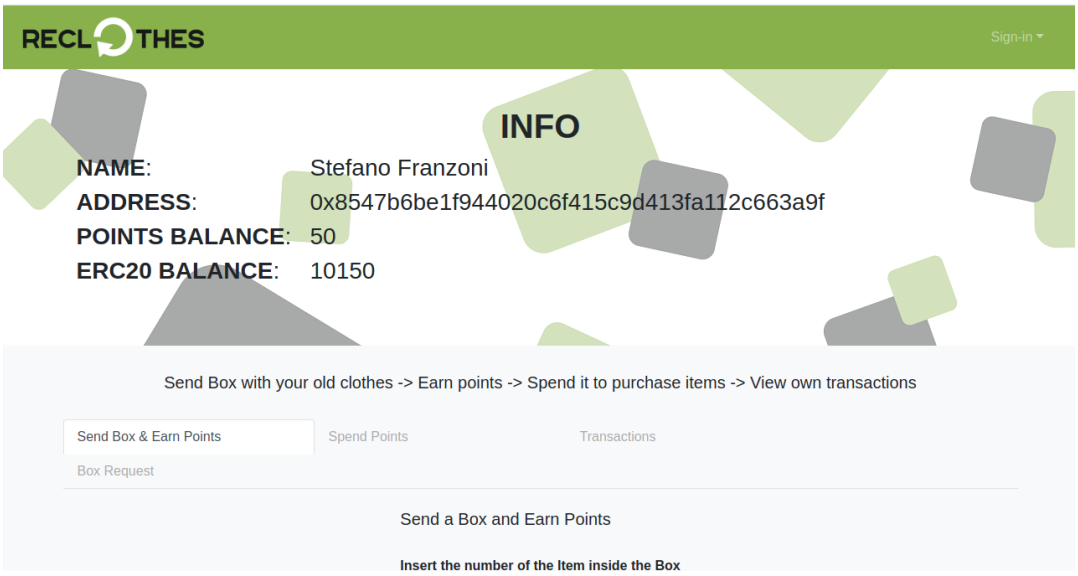

Figura 11. User Info

Reclothes Admin Page

In the previous sections we talk about a logical split about Admin for User and Admin for Producers. In the following views we divide the feature releated to the User type to be handled and there's a streight distinctions about Users and Producers.

Admin For Users This section show the view of the Admins that handle User side.

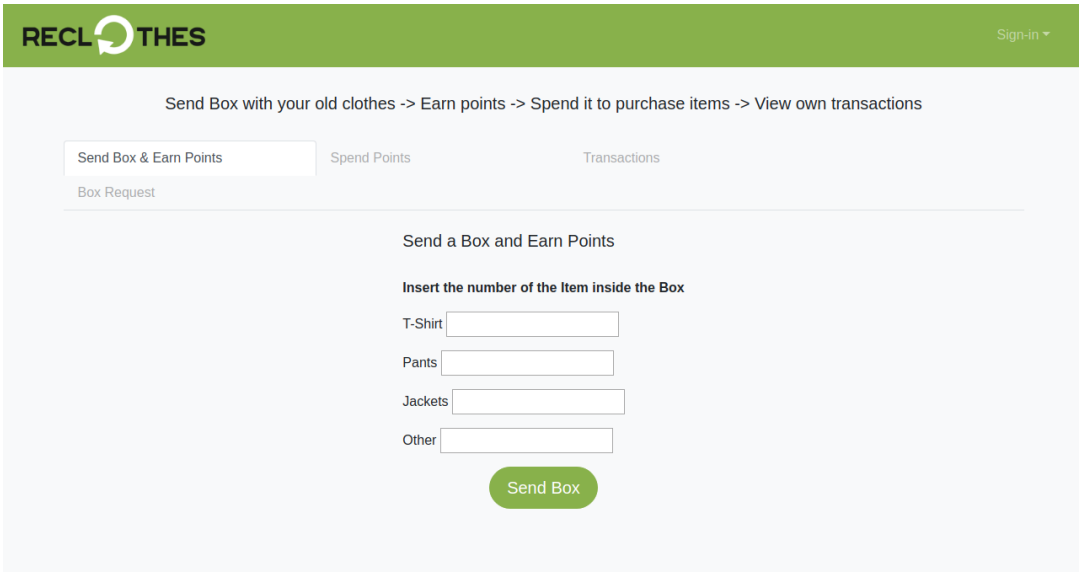

Figura 12. Send Box

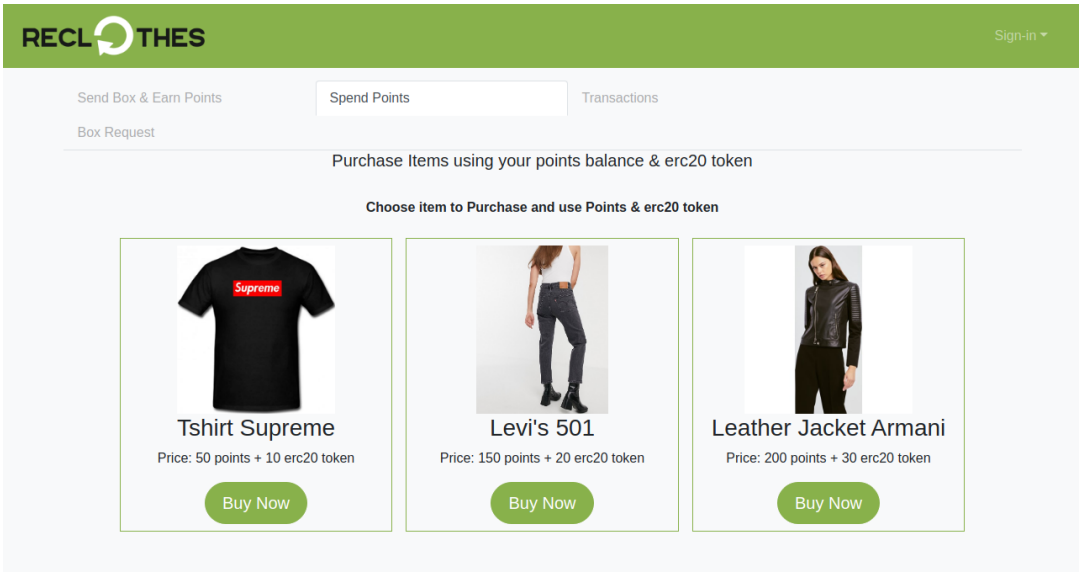

Figura 13. Purchase Clothes

Admin For Producers This section show the view of the Admins that handle Producer side.

Producer

This section show the view of the Producer side.

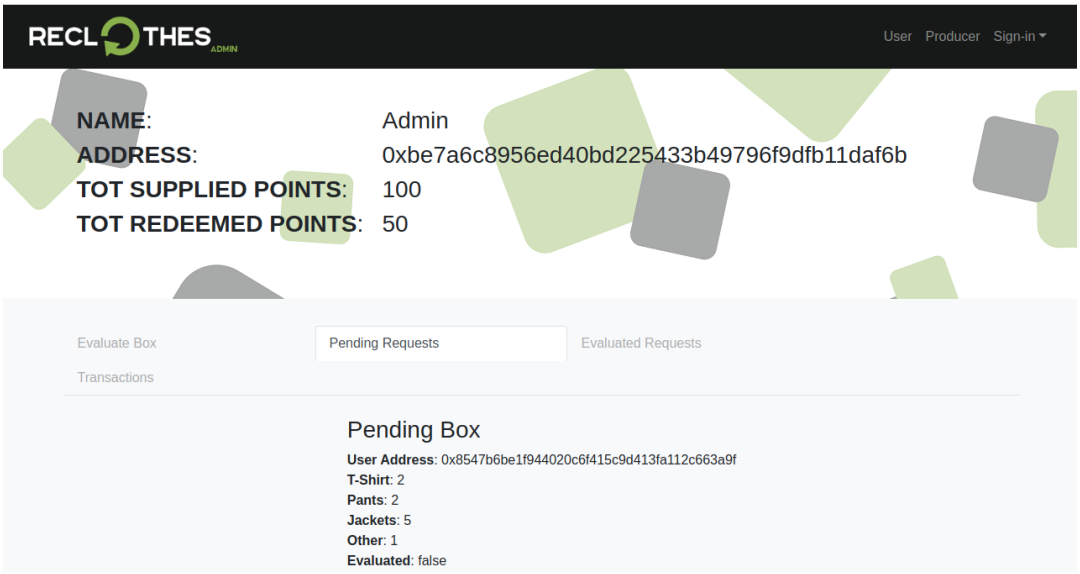

Figura 14. Admin Info

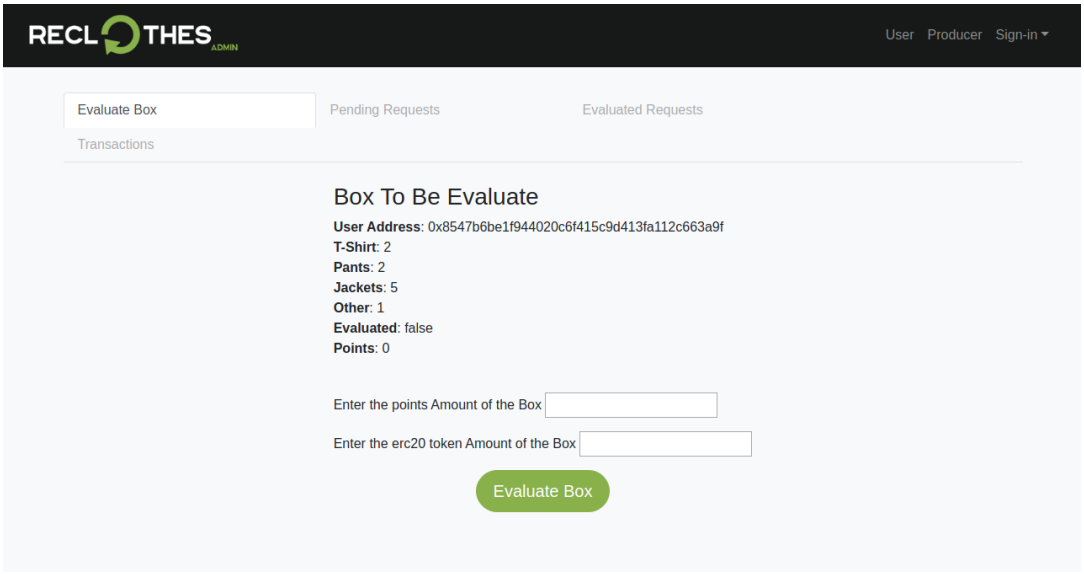

Figura 15. Evaluate Box

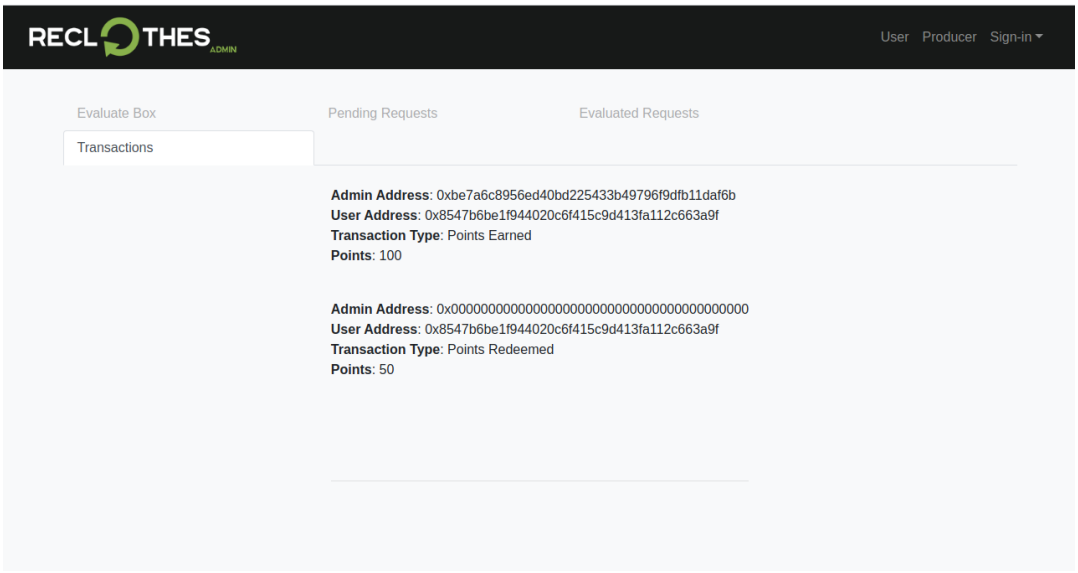

Figura 16. Transactions

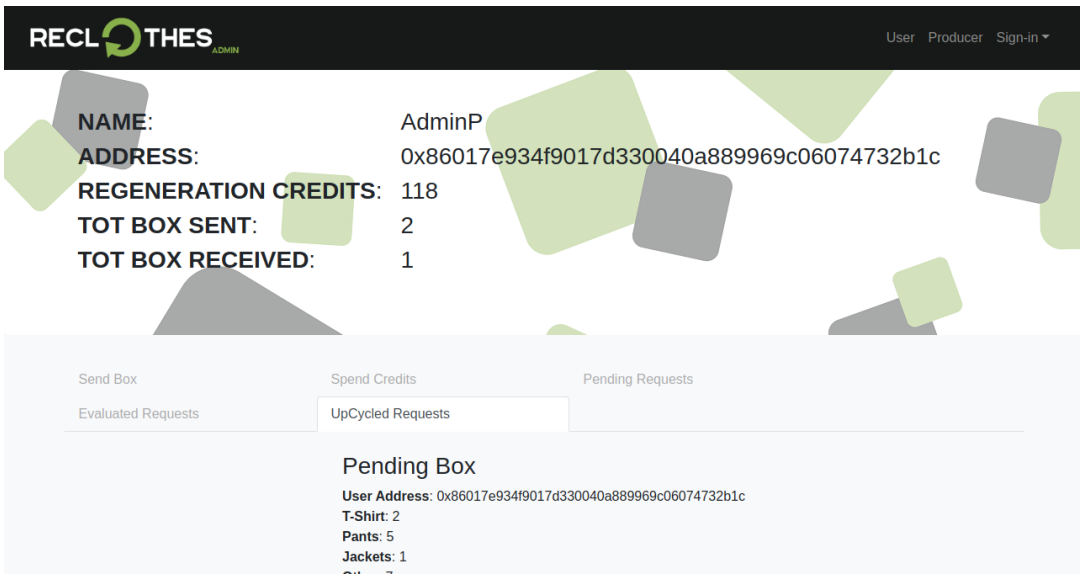

Figura 17. Admin for Producers Info

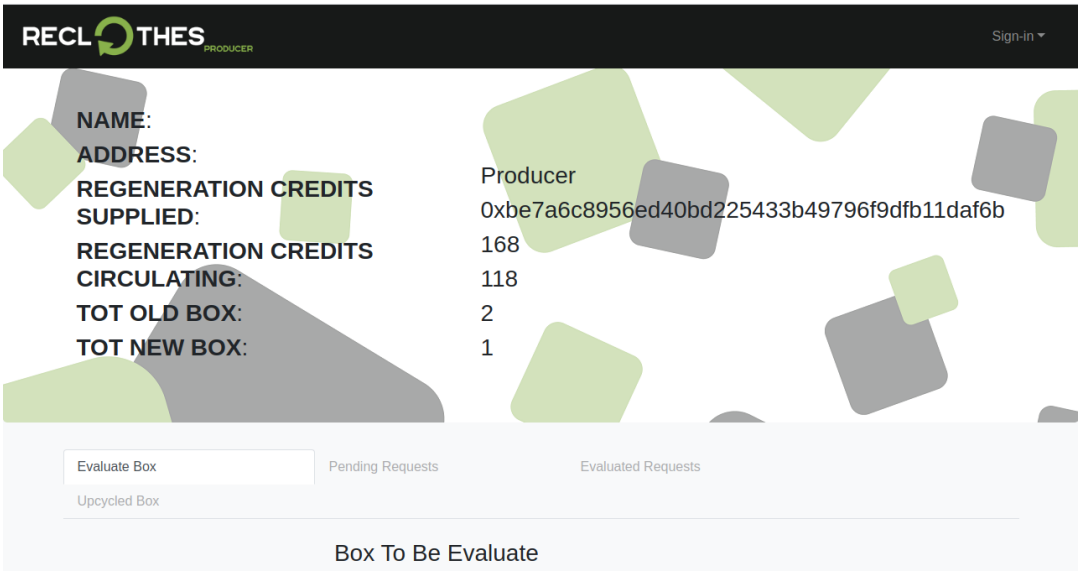

Figura 18. Producers Info