

Smart contract on Ethereum

Supervisor:

Dr. Mohammad Hossein Manshaei
Ms. Masoume Zare

Authors:

Arman Riasi
Ghazale Zehtab

June 2020

ITFundamentals

Introduction:	3
The main players and the rules of the smart contract:	3
The main benefit of the selected smart contract in comparison with traditional contracts:	4
Some details of the smart contract code:	6
The cryptocurrency in the smart contract:	9
Smart contract programming language:	10
References	10

Introduction:

Your data today is collected, monetized, and utilized without your consent or awareness. The data industry today is dominantly controlled by centralized services that aggregate data illegally and sell them to other enterprises to generate huge revenues, therefore your data privacy is being compromised. Also, there are no existing data marketplaces that allow enterprises to purchase high quality, insightful, and legally acquired data at reasonable prices for their business intelligence and targeted marketing purposes. Recent years have seen an increased emphasis on stricter regulation with regards to consumer data privacy and personal data ownership. Blockchain shares similar goals and motivations. Both aim at decentralizing data control and returning sovereignty of control from centralized entities back to the individual. Aside from sharing similar goals, blockchain can effectively help organizations, data-centric applications, and data owners with the GDPR and other data-related regulations. This is where Airbloc positions itself.

Airbloc Protocol redefines how data is collected, monetized, and utilized. Leveraging blockchain technology and token economics, it seeks to facilitate more transparent data flow between data owners, data providers, and data consumers. Ultimately, it aims to return data ownership back to data owners, provide applications with tools to collect and monetize data legitimately and allow data consumers to purchase explicitly consented data with an auditable source of provenance for their business intelligence, research, and targeted marketing purposes. "tokens" represent a diverse range of digital assets. In this way, tokens are essentially smart contracts that make use of the Ethereum blockchain. Airbloc offers two types of tokens: **Airbloc (ABL)** and **Airbloc Reward (AIR)**.

- ❑ ABL (Airbloc): It is mainly used as a means of participating in the network such as payment settlement by data consumers for data exchange and staking to register and maintain a node.
- ❑ AIR (Airbloc Reward): It is used primarily as a means of providing rewards to participants on the network.

In this review, we are going to consider the ABL.

The main players and the rules of the smart contract:

As mentioned earlier, the main players in this smart contract are:

- ❑ data owners,
- ❑ data providers,
- ❑ and consumers.

This smart contract is used as a means of payment, settlement, and participation in Airbloc. To purchase data, data consumers need to pay with ABL. To participate in the network and be rewarded for being a data provider, a data validator node, or an identity manager node, a certain amount of ABL is required to be staked as collateral. ABL tokens that data consumers paid for data are converted to AIR tokens and rewarded to data owners and data providers of the data via smart contracts.

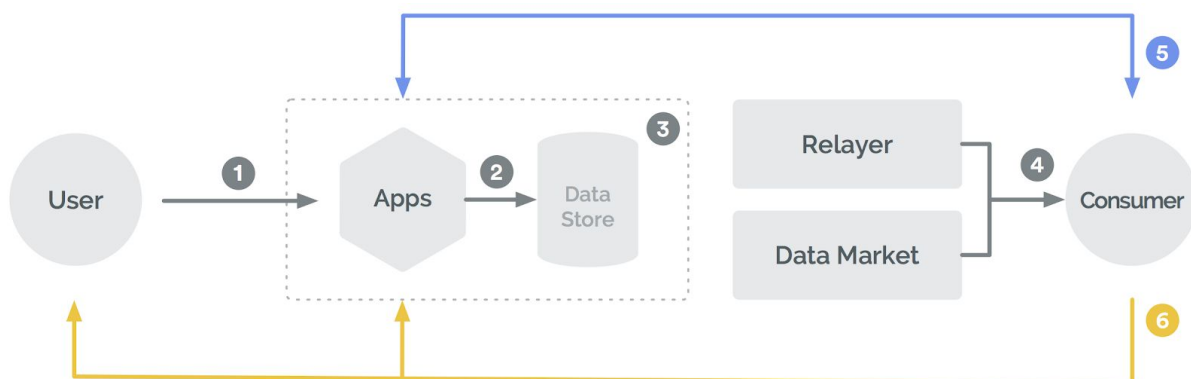
The main benefit of the selected smart contract in comparison with traditional contracts:

As we studied before there are 7 main differences between traditional contracts and smart contracts:

1. Third-party
2. Execution time
3. Remittance
4. Transparency
5. Archiving
6. Security
7. Cost

Now we're going to compare these parameters in the selected smart contract.

1. **Third-party:** This smart contract as others doesn't have a third party. One of the goals for migrating to the smart contract from a traditional contract was to delete the third party. This picture is a high-level overview of the data collection, registration, storage, and trading process in Airbloc, as you see there isn't a third party like a traditional contract.



2. **Execution time:** This smart contract is much faster than a traditional contract. This smart contract is about ownership. In a traditional contract for transferring or extending ownership, users must spend time going to another place, wait, and finally do what they have wanted. After some days the ownership would be changed, but this smart contract does it in just less than a minute.
3. **Remittance:** It's obvious that traditional contracts aren't automatic. The Airbloc has many smart contracts one of them is StandardToken and define like this:

```
contract StandardToken is ERC20, BasicToken {}
```

This smart contract has functions that make things automatic and far different from traditional ones. Functions are named and defined in the below:

- function transferFrom(address _from, address _to, uint256 _value) public returns (bool){} : Transfer tokens from one address to another
 - function approve(address _spender, uint256 _value) public returns (bool) {}: Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
 - function allowance(address _owner, address _spender) public view returns (uint256) {}: Function to check the amount of tokens that an owner allowed to a spender.
 - function increaseApproval(address _spender, uint _addedValue) public returns (bool) {}: Increase the amount of tokens that an owner allowed to a spender.
 - function decreaseApproval(address _spender, uint _subtractedValue) public returns (bool) {}: Decrease the amount of tokens that an owner allowed to a spender.
4. **Transparency:** The traditional contract transparencies are unavailable but in this smart contract they are public and available.
 5. **Archiving:** In the traditional contract archiving needs notes and papers and it's difficult but here by default, all personal data in Airbloc is stored directly by the data provider. In addition to personal data, Airbloc handles a variety of additional information (e.g. Data Registration Information, App Information, ...) called metadata which uses a meta-database for storage to ensure that only a minimum amount of information related to the smart contract or the proof of data is uploaded to the blockchain. Metadata is stored in BigchainDB for easier information retrieval and lower management cost. Currently, BigchainDB uses Practical Byzantine Fault Tolerance-based consensus¹² (PBFT); only permission operators can operate BigchainDB. This makes absolute architectural decentralization difficult. Therefore, the metadatabase is governed by Airbloc and its ecosystem partners.
 6. **Security:** In the traditional contract security isn't very high because a person or groups must take care of the data. They can save them in the lockbox or other safe things but their abilities are limited. Since only the metadata is stored in BigchainDB, it has no privacy-related risks. The hash value and the Merkle proof can be added to the blockchain to minimize using on-chain storage while maximizing security. Furthermore, once BigchainDB's decentralized structure is fully operating according to their roadmap, Airbloc will transfer the governance of the meta-database to the public so that individuals can maintain the metadatabase nodes and receive contribution rewards in a permission-less manner.
 7. **Cost:** In the traditional contract, clerks must be paid, energy is more used, other things like buildings are needed so in this smart contract cost is much lower. Databases, codes, and ... aren't as expensive as the traditional contract's requirements.

Some details of the smart contract code:

In the first line of your smart contract which is written by solidity, you need to use the following code.

```
pragma solidity ^0.4.19;
```

It tells you that the source code is written for Solidity version 0.4.19 or newer. **Pragmas** are common instructions for compilers about how to treat the source code.

Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables and functions that can modify these variables. A contract and its functions need to be called for anything to happen. To define a contract just need to use the following syntax and after that, you can define your functions and variable inside the curly bracket:

```
contract [the of contract] { }
```

When a contract is created, its constructor (a function declared with the **constructor** keyword) is executed once. A constructor is optional and only one constructor is allowed.

The following code is one of the smart contracts used in the Airbloc protocol. We'll explain all parts of this code in the following.

```
contract OwnableToken {  
  
    mapping (address => bool) owners;  
  
    event OwnershipTransferred(address indexed previousOwner, address  
indexed newOwner);  
    event OwnershipExtended(address indexed host, address indexed guest);  
  
    modifier onlyOwner() {  
        require(owners[msg.sender]);  
        _;  
    }  
}
```

```

    constructor() public {
        owners[msg.sender] = true;
    }

    function addOwner(address guest) public onlyOwner {
        require(guest != address(0));
        owners[guest] = true;
        emit OwnershipExtended(msg.sender, guest);
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        owners[newOwner] = true;
        delete owners[msg.sender];
        emit OwnershipTransferred(msg.sender, newOwner);
    }
}

```

Now let's explain the code precisely. In this code, first of all, define a contract and its name is OwnableToken. The constructor is the first part of the smart contract that is executed.

```

    constructor() public {
        owners[msg.sender] = true;
    }

```

msg.sender is the address currently interacting with the contract. It can be a human or another contract. So if a human is interacting with a contract, msg.sender is the address of the person. And if another contract (B) is interacting with the contract, the contract (B)'s address becomes msg.sender. If we want to have multiple owners in a smart contract we need to use mapping for storing them. Mappings act as hash tables which consist of key types and corresponding value type pairs. Mapping types are declared as `mapping(_KeyType => _ValueType) mappingName`. Here `_KeyType` can be almost any type except for a mapping, a dynamically sized array, a contract, an enum, and a struct. `_ValueType` can actually be any type, including mappings. This mapping is necessary for us to map the address to the corresponding owners.

```
mapping (address => bool) owners;
```

Consequently, the constructor sets the address of who is interacting with the contract 'true'. Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible. We have two events in this contract.

```
event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
```

The above event is for ownership transfer. It has two arguments, the first one is the address of the previous owner and the latter is the address of the new owner.

```
event OwnershipExtended(address indexed host, address indexed guest);
```

This event is for ownership extend. It has two arguments, the first one is the address of the host and the latter is the address of the guest.

```
modifier onlyOwner() {  
    require(owners[msg.sender]);  
    _;  
}
```

Modifiers create additional features on function or apply some restriction on function. The way to make a modifier is similar to the way to make a function. Instead of a keyword 'function', you could replace it to 'modifier' and insert underscore _ into modifier definition. A modifier decorates the function. It could decorate(execute) first before the original function, or after the original function, or else. Where to put underscore _ decides where to execute the original function. This modifier checks whether the address(msg.sender) is the same with the owner address defined on the contract. Therefore only the owner can execute these functions.

We have two functions in this contract. One of them is used for emitting the ownership extension and the second one is used for transferring the ownership.

```
function addOwner(address guest) public onlyOwner {  
    require(guest != address(0));  
    owners[guest] = true;  
    emit OwnershipExtended(msg.sender, guest);  
}
```

For ownership extension, the first line `require(guest != address(0));` is to check whether the target account is the zero-account (the account with the address 0), the transaction creates a new contract. The address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the “nonce”). After that, it makes the guest address true. At the end emit `OwnershipExtended` event with two parameters, `msg.sender` as the host and the second parameter as a guest. Emit is used to emit any events.

```
function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    owners[newOwner] = true;
    delete owners[msg.sender];
    emit OwnershipTransferred(msg.sender, newOwner);
}
```

As mentioned earlier, This function is for transferring the ownership. The first line `require(newOwner != address(0));` is to check whether the target account is the zero-account or not. After that, it makes the new owner address true in the mapping list. Then, delete the previous owner from the mapping list. Finally, emit the events `OwnershipTransferred` with two parameters. `Msg.sender` as previous owner and the second parameter as new owner.

The cryptocurrency in the smart contract:

As explained in the security part of the comparison between traditional and smart contracts the Airbloc Protocol (also its smart contracts) are using HASH and blockchain.

In **Private Identity Matching**, Airbloc uses zero-knowledge proof to prevent the exposure of sensitive identity information such as PIID or DIID during the identity matching process. Therefore, identity matching is performed by the following procedures:

1. Alice uses the arbitrary value r as Salt and computes h which is the hash of the identity information i to match.
2. Alice propagates h and r to the Identity Manager Network.
3. Bob, an identity manager which has the same i_{Bob} as i , verifies that hash i_{Bob} matches h that Alice has propagated. Confirms, $\text{hash}(i_{Bob} + r) = h$.
4. Bob sends i_{Bob} to Alice via a private channel, which certifies Alice has the identity i to match.
5. Bob returns the AID, not the actual user ID, as a result of the match.
 $uA = \text{AID}(u)$
6. Bob does not just return uA to Alice, but reveals a zero-knowledge proof π that $\text{hash}(u)$ value corresponds to uA . However, for the above process the value of $\text{hash}(u)$ must be registered on the blockchain in advance.
7. Alice proves π , and when it succeeds, identity matching is accomplished.

Simply put, this process ensures that there is indeed the existence of valid identifiers by the requestor by sending the hashed version of those identifiers to the verifiers.

In **DAuth Process**, Airbloc records the authorization on the blockchain:

- Applications can only monetize data which users authorized.
- Unpermitted data will be filtered during the data cleansing process.

In **Temporary Account Creation**:

1. Airbloc hash-locks the account on the blockchain.
2. To unlock the account (sign-in into & turn it from temporary to the active account), the user must type in the original ID i to unlock the hashlock.
 - a. To further protect the user's privacy, double-hashing of the original ID can be considered.

In **Data Exchange Process** (smart contract of the previous part), after the transaction is completed, the data exchange record is stored on the blockchain and data owners can be notified of their data flows.

As an example in the codes of the smart contract in the previous part this line is doing guess and try for hashing:

```
require(guess != address(0));
```

Smart contract programming language:

As you can see the Airbloc uses Ethereum. Hence the smart contracts are written by Solidity language.

References

[1] <https://airbloc.org/>

[2] <https://github.com/airbloc/token/blob/master/contracts/ABL.sol>