FuturICT 2.0 Challenge 1 - Token Creator

Team “E-Mine 1” project documentation

Blockchain and Internet of Things School (BIOTS) 2018
12.2. - 16.2.2018., Zurich, Switzerland

Team “E-Mine 1” members:
Andrin Bertschi
Nenad Katanic
Matija Piskorec
Julia Füreder
Vikas Patil

Code repositories:
https://github.com/e-mine1/e-mine-web
https://github.com/e-mine1/GenesisApp

Outline of the documentation

This document summarizes the contributions of the team “E-Mine 1” on a hackaton organized as a part of Blockchain and Internet of Things School (BIOTS) 2018. Major contributions are:

- Development of the E-Mine Token generator [1] - a web service for creation and deployment of Ethereum smart contracts.
- Extension of the GenesisApp [2] - an Android application for easy generation of custom token contracts. We extended the GenesisApp with several standard token templates, and implemented the compilation and deployment of tokens by requesting it through the E-Mine Token Generator web service.

In order to showcase the utility of our E-Mine Token generator we used it as a part of the token-based solution for incentivizing e-waste recycling. This use case was developed in collaboration with the “E-Mine 2” team [3] which participates in FuturICT 2.0 Challenge 2 - Token obtainer.

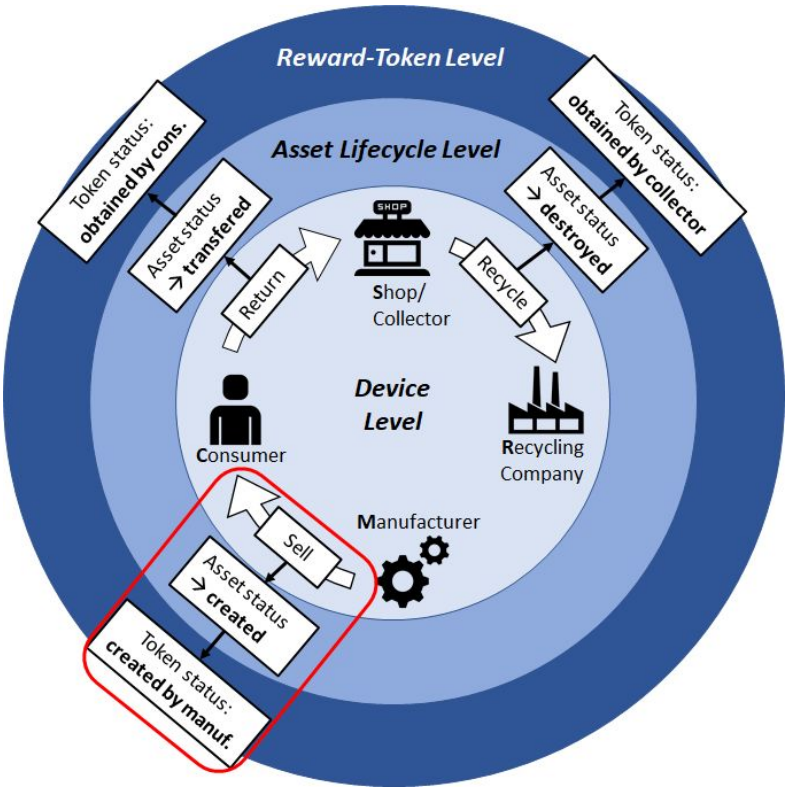
[1] E-Mine Token Generator, https://github.com/e-mine1/e-mine-web
[2] Fork of the GenesisApp, https://github.com/e-mine1/GenesisApp
[3] E-mine2 Github organization, https://github.com/e-mine2

Introduction

Electronic waste recycling is currently a huge problem the world faces today. Electronic devices become e-waste as soon as they are at the end of their useful life, or in some cases when they are no longer wanted because they become obsolete over time. These discarded products often contain hazardous amounts of chemicals, but also they are partly made out of rare metals which gives them a certain value. By now consumers of electronics devices do not benefit from returning their products, so they also have no incentives to do recycling after utilization. Furthermore there is no lifetime tracking of these products, what finally leads to no data whether the products are recycled or not. We have to deal with increasing amounts of old electronic devices, which is also a result of the shortened life cycle of each device. The aim of this project is to incentivise consumers to finally recycle their products after usage by creating a blockchain based network. The idea is to reward the consumer in return.

The Solution

The current project aims for a solution to incentivize tracking of electronic devices and increase their recycling. This is attempted by implementing a blockchain network, described as follows.



To incentivize various stakeholders to track and recycle electronic devices, a blockchain-based network that rewards Tokens could be implemented. We believe that the manufacturers would have incentive to obtain the lifecycle history of their devices (for various reasons) tracked by the blockchain. In exchange for this tracking info, they would be willing to finance reward tokens.

1. When a device is manufactured, an asset is created.
2. This asset is tracked by the blockchain as it traverses through the system, from manufacturer to distributor to consumer to collector and ultimately to the recycler.
3. When the consumer returns the device to a collection center (or shop authorized to collect returned devices), a reward token is obtained by the consumer.
4. When the collector hands over the device to the recycler, another reward token is also granted to the shop.
5. When the recycler proves that the device was recycled, this info is transferred to the manufacturer.
6. In reality, there is no direct material link between the recyclers and the manufacturers because they are usually based in different continents, or the quality of recycled raw material may not be acceptable for the manufacturer.
7. So in effect, the current solution creates incentives in the system to track and return devices for recycling in the form of tokens and valuable lifecycle information.

Tokens. Tokens are commonly used to represent a particular asset or utility. They usually reside on top of another cryptocurrency blockchain (Ethereum in our case). Tokens can represent basically any assets that are fungible and tradable: commodities, loyalty points and even other cryptocurrencies. When creating tokens we do not have to modify the underlying code (logic) from a particular protocol/blockchain or create another blockchain from scratch as it is the case with alternative cryptocurrencies (also known as altcoins). All we have to do is to conform to a standard template on the existing blockchain – such as on the Ethereum platform – that allows us to create our own tokens. This token creation functionality is made possible through the use of *smart contracts concept*.

Smart contracts. A smart contract (also referred to as a token contract in the context of token creation) is a piece of code that maps account addresses to their balances. Balance is actually a value defined by contract creator (owner) and can represent different assets (physical objects, reputation, monetary value etc.). The unit of this balance is called a token.

Total supply of tokens can be:

- Increased - by *minting* new tokens (producing them for the first time)

- Decreased - by *burning* tokens

ERC-20 standard [4] came about as an attempt to provide a common set of features and interfaces for token contracts in Ethereum [5]. ERC-20 has many benefits, such as allowing wallets to provide token balances for hundreds of different tokens and creating a means for exchanges to list more tokens by providing nothing more than the address of the token's contract. Currently most of active token contracts are ERC-20 compliant tokens. An ERC-20 token contract is defined by the *contract's address* and the *total supply* of tokens available to it, but has a number of optional items that are usually provided as well to provide more detail to users. These are the *token's name*, its *symbol*, and the *number of decimals*.

E-Mine Token Generator web service

The problem. Creation of smart contracts is still laborious task requiring specialized knowledge. Technically, it involves - 1) specifying a smart contract in Solidity or some other appropriate language, 2) compiling it into bytecode and 3) deploying it on the Ethereum network. Properties of a smart contract can be specified on any convenient user interface such as a mobile app or a web page. An actual smart contract can then be prepared by filling a template of a smart contract. On the other hand, compiling and deployment of smart contracts is not readily available on all platforms. As For example, currently there no Java based library for compiling the Solidity code [6], making it impossible to compile (and deploy) the smart contract directly through the Android mobile application such as GenesisApp [7].

Our solution. We decided to build E-Mine Token Generator [8] web service through which users can submit requests for creation and deployment of their smart contracts. We built a web service using Flask [9]- a widely used Python [10] based web framework. Web service can be accessed through a REST API. User makes a request for a smart contract and provides relevant parameters exposed through the user interface. The web service then prepares an actual smart contract by filling a template of Solidity code, compiles it and broadcasts it on Ethereum network. e-mine-web provides compilation and deployment of several standard tokens which are available in OpenZeppelin framework [11] (Table 1.). For compilation and deployment on the server side we used Truffle [12] - a Node.js [13] based development framework

---

[4] https://theethereum.wiki/w/index.php/ERC20_Token_Standard
[5] https://medium.com/@jgm.orinoco/understanding-erc-20-token-contracts-a809a7310aa5
[6] As of February 2018.
[7] GenesisApp, https://github.com/FuturICT2/GenesisApp, https://github.com/FuturICT2/Genesis
[8] e-mine-web , https://github.com/e-mine1/e-mine-web
[9] Flask, http://flask.pocoo.org/
[10] Python, https://www.python.org/
[11] OpenZeppelin, https://openzeppelin.org/
[12] Truffle, http://truffleframework.com/
[13] Node.js, https://nodejs.org/en/

for Ethereum smart contracts. In order to test our system we set up a local Ethereum network using Ganache [14], a tool which is a part of the Truffle framework. Ganache features an internal Javascript implementation of Ethereum blockchain, as well as built-in blockchain explorer that allows easy testing and debugging of distributed applications (Figure 2.). The overall system architecture of the E-Mine Token Generator is displayed on Figure 1.

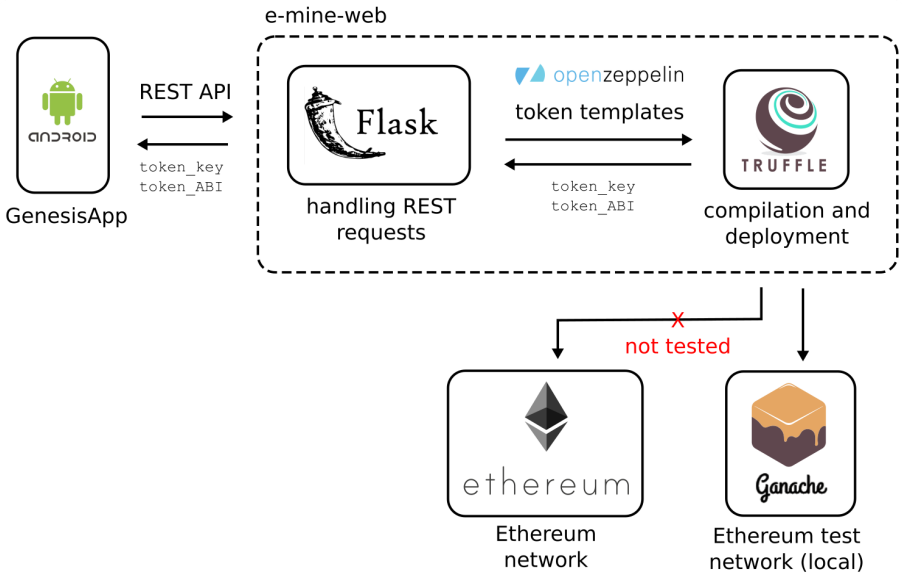


Figure 1. Architecture of the E-Mine Token Generator web service for compiling and deploying the smart contracts to Ethereum network.

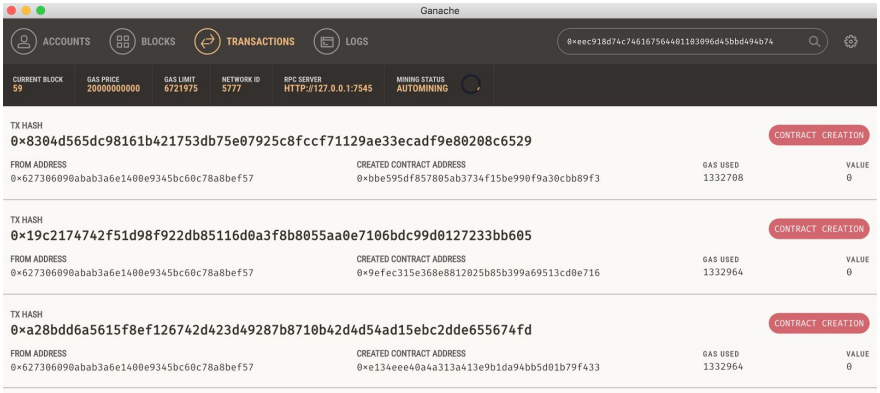


Figure 2. A screenshot of Ganache's built-in explorer showing couple of transactions generated by the E-Mine Token Generator service on a test Ethereum network.

Security concerns. We should emphasise that our solution is currently in a prototype phase and should not be used in a production environment. Largest security concern, which we do not address at all, is the management of public and private keys. Currently, the client sends its private key directly to web service and the service is responsible for compilation and

[14] Ganache, http://truffleframework.com/ganache/

deployment in the name of the client. This exposes client to the potential fraud by the web service provider, as well as by any malevolent eavesdropper in the communication channel. Key management is one of the functionalities which could be implemented in the future (section Future work).

Smart contract templates. As a basis for token creation we use OpenZeppelin, a framework to build secure smart contracts on Ethereum. For each token type a template is provided, implemented as an extended class of corresponding OpenZeppelin token implementation. Following token templates are provided:

Token Standard	Token	Description
ERC20	Basic Token	Implementation of the basic standard token
	Burnable Token	Standard token with no allowances
	Capped Token	Standard token modified with pausable transfers
	Mintable Token	A Simple ERC20 with mintable token creation
	Pausable Token	Mintable token with a token cap
	Standard Token	Token that can be irreversibly burned (destroyed)
ERC721	Generic ERC721 Token	Generic implementation for the required functionality of the ERC721 standard
ERC827	Generic ERC827 Token	ERC827 implementation - ERC20 standard with extra methods to transfer value and data and execute calls in transfers and approvals

Table 1. Token templates available on our web service. All token templates have the following properties exposed: name, symbol, decimals, initial supply.

Each template contains placeholders for custom user parameters provided by the client application. Currently, the following parameters are supported:

- *Token name* - Name of a token.
- *Token symbol* - Symbol representing the token.
- *Decimals* - Divisibility of a token, i.e. number of decimal places when displaying token value.
- *Initial supply* - Initial supply of tokens.

Each template can be easily extended to introduce new smart contract (token) parameters and functions according to the specific user requirements.

Prior to integrating manually written templates with the rest of the E-Mine Token Generator system, we tested token creation (for all token types supported by our generator) using the Remix Solidity IDE. Figure 3. shows generation Following paragraphs refer to Standard token type as a showcase example.

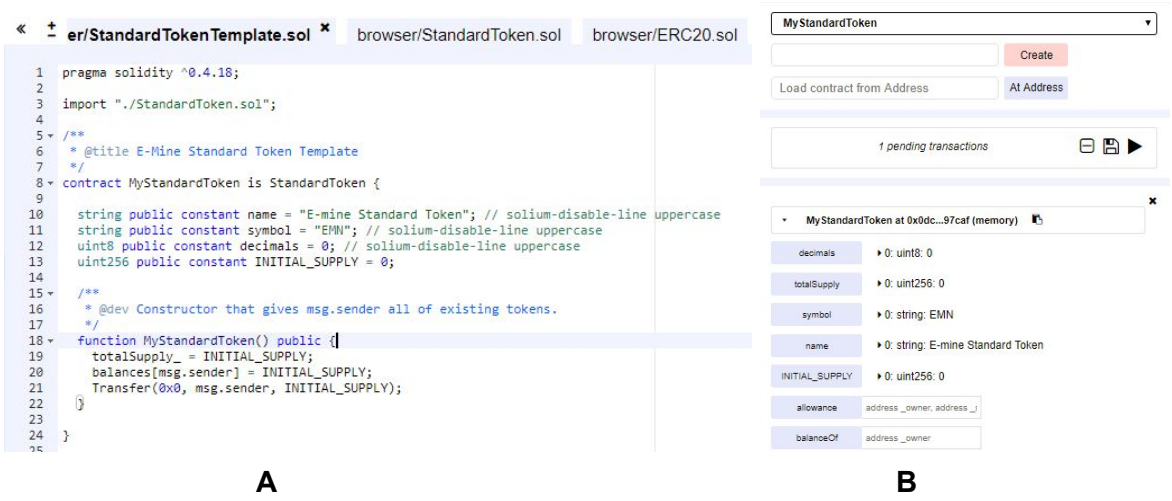


Figure 3. A smart contract is implemented as an extended class of the provided OpenZepellin smart contract (panel A). In our MyStandardToken contract we introduce additional parameters (name, symbol, decimals and initial supply). We make sure that newly introduced parameters hold and return correct values after successful creation (panel B).

As a final step, we remove hardcoded values, and introduce the corresponding placeholders that will be updated with real values provided by the Token Generator Web service during the creation of new token:

```
string public constant name = "%token_name%";
string public constant symbol = "%token_symbol%";
uint8 public constant decimals = %token_decimals%;
uint256 public constant INITIAL_SUPPLY = %token_initial_supply%;
```

We then used the same approach to create templates for all other token types, as listed in Table 1.

REST API. E-Mine Token Generator exposes following REST API:

API URL	Method	Required/Optional variables	Expected response
/api/requests/	GET	id=[integer]	
/api/tokens/types	GET		{ "types": [

			"MyBasicToken", "MyBurnableToken", "MyCappedToken", "MyERC721Token", "MyERC827Token", "MyMintableToken", "MyPausableToken", "MyStandardToken"] }
/api/tokens/create	POST	tokenName=[string] symbol=[string] maxSupply=[integer] decimals=[integer] genesisSupply=[integer]	{ "created": [datetime], "key": [uuid], "status": "success" or "pending" or "failure", "token_abi": [base64 encoded JSON value] "token_addr": "0xaba7902442c5739c6f0c182691d48d63d06a212e", "updated": [datetime], "version": 1}

Table 2. REST API endpoints for the E-Mine Token Generator web service.

Installation instructions

First, install all Python prerequisites required for the web server. They are listed in requirements.txt file and can be easily installed with pip [15]:

```
$ pip install -r requirements.txt
```

Second, install Truffle which is needed for compilation and deployment of smart contracts. It can be installed with Node package manager [16]:

```
$ npm install -g truffle
```

Future work

There are several directions for future work and potential upgrades of the E-Mine Token Generator.

Key management. Implement secure communication between the client and the web service.

Web-based user interface. Use a web-based user interface instead of an Android app.

[15] pip, https://pypi.python.org/pypi/pip
[16] npm, https://www.npmjs.com/

Compilation and deployment of smart contracts on client side. Implement compilation and deployment of smart contracts on a client side instead of using a web service.

Deployment on Ropsten test network. Deploy smart contracts on the Ropsten test network instead of a local network generated by Ganache.