Non-Fungible Token Standard on EOS

Non-Fungible Token Standard on EOS

Author	Defibox Official
Type	Stansard Track
Category	EOS NFT
Create	2021-09-23

Table of Contents

Summary

A standard interface for non-fungible tokens, base on pink.network's work.

Motivation

A standard interface allows wallet/DeFi to work with any NFT on EOS. As NFT is getting more and more popular and people are increasingly recognizing on-chain collectibles, We bring out this standards for everyone who want to issue their NFT can refer this standard and can compatible with all platforms on EOS.

Standard Interface

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

We define all the basis standard interfaces that all NFT must implement. Every NFT contract which compatible with this standard must implements the following interface.

```
#include <eosio/eosio.hpp>
#include <eosio/singleton.hpp>

```

```
using namespace eosio;
using namespace std;

/*
@title EOS Non-Fungible Standard
*/

CONTRACT nftstandard : public contract{
 // types
 public:

 /*
 @dev Store data type that be used in serialization and deserialization
 @param name The field name
 @param type The data type
 */
  struct FORMAT {
      string name;
      string type;
  };

  /*
    @dev Define schema that be used in serialization and deserialization
    @param schema_name The schema name
    @param format Vector of the format
  */
  struct schemas_s {
    name            schema_name;
    vector <FORMAT> format;

    uint64_t primary_key() const { return schema_name.value; }
  };

  /*
  @dev Store the overall information of each NFT asset
  @param collection_name Name of the collection name
  @param author The author of this collection
  @param allow_notify If this NFT need to notify other accounts
  @param authorized_accounts  Accounts can create new templates of this collection, mint assets of this collection and edit the mutable data of templates/ assets of this collection.
  @param market_fee Fee out of every asset sale of this collection and make it available to the collection's author
  @param serialized_data Serialized data of collection info.
  */
   struct collections_s {
    name             collection_name;
    name             author;
    bool             allow_notify;
    vector <name>    authorized_accounts;
    vector <name>    notify_accounts;
    double           market_fee;
    vector <uint8_t> serialized_data;

    uint64_t primary_key() const { return collection_name.value; };
    };

```

```
    /*
    @dev Store duplicate data of NFTs
    @param sheam_name The schema that use for serialization and deserialization
    @param transferable If that NFT is transferable
    @oaram burnable If that NFT is burnable
    @param max_supply Max supply of the NFT asset
    @param issued_supply The issued NFT amount
    @param immutable_serialized_data The immutable data that blong to all NFT asset of that collection
    */
    struct templates_s {
        int32_t          template_id;
        name             schema_name;
        bool             transferable;
        bool             burnable;
        uint32_t         max_supply;
        uint32_t         issued_supply;
        vector <uint8_t> immutable_serialized_data;

        uint64_t primary_key() const { return (uint64_t) template_id; }
    };

    /*
    @dev Record the asset info
    @param collection_name The collection this asset belongs to
    @param schema_name The scheam format that the data should follow for serizlization/deserialization
    @param template_id The templeate is belongs to
    @param immutable_serialized_data Data will not be changed
    @param mutable_serizlized_data Data that will be changed
    */

    struct assets_s {
        uint64_t         asset_id;
        name             collection_name;
        name             schema_name;
        int32_t          template_id;
        name             ram_payer;
        vector <uint8_t> immutable_serialized_data;
        vector <uint8_t> mutable_serialized_data;

        uint64_t primary_key() const { return asset_id; };
    };

    /*
    Types that use to asset and template data serialization
    */
    typedef std::vector <int8_t>      INT8_VEC;
    typedef std::vector <int16_t>     INT16_VEC;
    typedef std::vector <int32_t>     INT32_VEC;
    typedef std::vector <int64_t>     INT64_VEC;
    typedef std::vector <uint8_t>     UINT8_VEC;
    typedef std::vector <uint16_t>    UINT16_VEC;
    typedef std::vector <uint32_t>    UINT32_VEC;
    typedef std::vector <uint64_t>    UINT64_VEC;
    typedef std::vector <float>       FLOAT_VEC;
```

```
    typedef std::vector <double>      DOUBLE_VEC;
    typedef std::vector <std::string> STRING_VEC;

    typedef std::variant <\
        int8_t, int16_t, int32_t, int64_t, \
        uint8_t, uint16_t, uint32_t, uint64_t, \
        float, double, std::string, \
        INT8_VEC, INT16_VEC, INT32_VEC, INT64_VEC, \
        UINT8_VEC, UINT16_VEC, UINT32_VEC, UINT64_VEC, \
        FLOAT_VEC, DOUBLE_VEC, STRING_VEC
    > ATTRIBUTES;

    typedef std::map <std::string, ATTRIBUTES> ATTRIBUTE_MAP;
    typedef multi_index <name("collections"), collections_s> collections_t;
    typedef multi_index <name("schemas"), schemas_s> schemas_t;
    typedef multi_index <name("templates"), templates_s> templates_t;
    typedef multi_index <name("assets"), assets_s> assets_t;


  nftstandard(name receiver, name code, datastream<const char*> ds) :
  contract(receiver, code, ds),
  {}



  //ACTIONS

  /*
  @dev Transfer NFT token
  @param from The Non-Fungible Token sender
  @param to The Non-Fungible Token receiver
  @param asset_id The tokenId to transfer
  @param memo The Remark information
  */
  ACTION transfer(name from, name to, vector<uint64_t> asset_id, std::string memo);

  /*
  @dev Create a template for a set of NFT asset
  @param authorized_creator Authorized operator that can creates the template for
  specific collection, should check the permission according authorized_accounts defines in collection table
  @param collection_name The collection name that this template belongs to
  @param sechema_name The schema name that defines how to serialize and deserialize data, should be same as the schema_name defines in collection table
  @param transferable If the NFT asset can be transferred
  @param burnable If the NFT asset can be burned
  @param max_supply The max supply of the NFT asset
  @param immutable_data The immutable data that all NFT assets blong to that template will inherit
  */

  ACTION createtempl(
    name authorized_creator,
    name collection_name,
    name schema_name,
    bool transferable,
```

```
    bool burnable,
    uint32_t max_supply,
    ATTRIBUTE_MAP immutable_data
    );

  /*
  @dev Create sechema that defines how data serialize and deserialize
  @param authorized_creator Authorized operator that can creates the schema for
  specific collection, should check the permission according authorized_accounts defines in collection table
  @param collection_name The collection name that this schema belongs to
  @parma schema_name The schema_name
  @param sehema_format Vector of the FORMAT type data
  */
  ACTION createschema(
    name authorized_creator,
    name collection_name,
    name schema_name,
    vector <FORMAT> schema_format
    );

  /*
  @dev Mint NFT asset
  @param authorized_minter Authorized operator that can mint NFT asset for
  specific collection, should check the permission according authorized_accounts defines in collection table
  @param collection_name The collection name that this NFT asset belongs to
  @param schema_name The schema name that defines how to serialize and deserialize data, should be same as the schema_name defines in collection table
  @param template_id The template that use to create NFT assets
  @param new_asset_owner The owner of that NFT
  @param immutable_data The immutable data of that asset
  @param mutable_data The mutable data of that asset
  */
  ACTION mintasset(
    name authorized_minter,
    name collection_name,
    name schema_name,
    int32_t template_id,
    name new_asset_owner,
    ATTRIBUTE_MAP immutable_data,
    ATTRIBUTE_MAP mutable_data,
    );

  /*
  @dev Create a NFT collection.
  @param author The author of this collection
  @param collection_name The name of the collection
  @param allow_notify The accounts want to notify when things happen
  @param authorized_accounts The authorized accounts that can operate mint, create schema and create template
  @param market_fee The sell and buy fee
  @param data Data that defines the info of that NFT, and is serialized follows the schema defined in that collection
  */
  ACTION createcol(
```

```
    name author,
    name collection_name,
    bool allow_notify,
    vector <name> authorized_accounts,
    vector <name> notify_accounts,
    double market_fee,
    ATTRIBUTE_MAP data
    );

};

/*
@dev To modify the mutable data of assets
@param authorized_editor Authorized editor that can modify the asset mutable data
@param asset_owner The owner of that asset
@param asset_id The asset id that want to modify
@param new_mutable_data The data to be modified
*/
 ACTION atomicassets::setassetdata(
    name authorized_editor,
    name asset_owner,
    uint64_t asset_id,
    ATTRIBUTE_MAP new_mutable_data
)

  /*
  @dev Burn the asset
  @param asset_owner The owner of the asset
  @param asset_id The asset id
  */
  ACTION atomicassets::burnasset(
    name asset_owner,
    uint64_t asset_id
);

  /*
  @dev Add accounts to notify, only author can do this
  @param collection_name The collection that needs to notify accounts
  @parma account_to_add The account name that will receive nofitiaction
  */
  ACTION atomicassets::addnotifyacc(
    name collection_name,
    name account_to_add
);

  /*
  @dev Remove the notified accounts, only author can do this
  @param collection_name The collection name that no longer notify accounts
  @param account_to_remove Account that removes
  */
  ACTION atomicassets::remnotifyacc(
    name collection_name,
    name account_to_remove
);

  /*
```

```
  @dev Add authorized accounts for a collection, only author can do this
  @param collection_name The collection name
  @param account_to_add The authorized account to add
  */
  ACTION atomicassets::addcolauth(
    name collection_name,
    name account_to_add
);

  /*
  @dev Remove the authorized accounts for a collection, only author can do this
  @param collection_name The collection name
  @param account_to_remove Account that removes
  */
  ACTION atomicassets::remcolauth(
    name collection_name,
    name account_to_remove
);

  ACTION logmint(
    uint64_t asset_id,
    name authorized_minter,
    name collection_name,
    name schema_name,
    int32_t template_id,
    name new_asset_owner,
    ATTRIBUTE_MAP immutable_data,
    ATTRIBUTE_MAP mutable_data,
    ATTRIBUTE_MAP immutable_template_data
    );

  ACTION atomicassets::logburnasset(
    name asset_owner,
    uint64_t asset_id,
    name collection_name,
    name schema_name,
    int32_t template_id,
    vector <asset> backed_tokens,
    ATTRIBUTE_MAP old_immutable_data,
    ATTRIBUTE_MAP old_mutable_data,
    name asset_ram_payer
);

  ACTION atomicassets::lognewtempl(
    int32_t template_id,
    name authorized_creator,
    name collection_name,
    name schema_name,
    bool transferable,
    bool burnable,
    uint32_t max_supply,
    ATTRIBUTE_MAP immutable_data
);

  ACTION atomicassets::logtransfer(
    name collection_name,
```

```
    name from,
    name to,
    vector <uint64_t> asset_ids,
    string memo
);

  ACTION atomicassets::logsetdata(
    name asset_owner,
    uint64_t asset_id,
    ATTRIBUTE_MAP old_data,
    ATTRIBUTE_MAP new_data
)

```

Caveats

Compiler

This standard is compiled in eosio.cdt >= 1.7.0, which is a stable version.

Rationale

NFT Identifiers

Every NFT is identified by a unique uint64_t ID inside the smart contract. This identifying number SHALL NOT change for the life of the contract. The pair (contract address, uint64_t tokenId) will then be a globally unique and fully-qualified identifier for a specific asset on an EOS. While some smart contracts may find it convenient to start with ID 0 and simply increment by one for each new NFT, callers SHALL NOT assume that ID numbers have any specific pattern to them, and MUST treat the ID as a “black box”. Also note that a NFTs MAY become invalid (be destroyed). Please see the enumerations functions for a supported enumeration interface.

Transfer Mechanism

When transfer the NFT tokens, the transfer function should notify both the sender and receiver via `require_recipient`, in order that contracts can make some actions when receives the NFT. Moreover. Transfers must be initiated by the owner of NFT. The transfer function also support multi tokenId transfer and the tokenId must exist.

Each contract that wants to receive NFT MUST implements an `on_nft_received` function so that the contract can act some actions with the custom logic inside the function.

Metadata

Each NFT has its own propertites. Different from Ethereum, in EOS, we store the metadata on chain, so that contracts can get properties of each NFT. But in this way it will bring about the problem of RAM usage, to solve this problem, we use the solution of atomic asset. In this way, attritube that passed into the asset are all in `ATTRIBUTE_MAP` type(defines below) and finally be serialized via protobuf. we will cut the metadata into `Template`, `Collection` and NFT own asset data. Here's the detail:

Collection

Collections group assets, schemas and templates together and manage the permissions for those. It's a set of NFT asset of same kind. like Cryptopunks on Ethereum. In EOS, we can refer the [atomic asset example](). They also have a data field that follow a unified and immutable schema meant specifically for collection infos.

Collections defines the basic attributes of a collection, including the market_fee, its author, accounts that need to be notified and authorized operate account and its name. The immutable_data file in the table is used to store Description info, and are serialized in a certain format. Every collection follows that format. Currently there is not a standard format but we suggest to use format that atomic assets already in use

The scope of the collection table MUST use the collection name as its scope, so that each collection can be unique.

Template

Template, as the name suggests, is template for creating batch of assets. Assets that follow a same template inherit all the properties that template have.

Templates' main purpose is to save RAM costs by storing data that is duplicate in a lot of similar assets only once. It is however also possible to define a max_supply within a template, in which case it could be used to group together assets with provable scarcity.

The scope of the template MUST use its collection name as its scope, so that one can query all template belongs to one collection for convenience

Schema

Schema is a set of data type that third party and contract use to serialize and deserialize data stored in `Template` and `Asset` and MUST never change. Follow the atomic asset standard, all data are serialize via protobuf, which is RAM cost saving.

Cause the `Schema` never change, so the third party can download the schema once and deserizlize the onchain data forever with that schema.

Scope of schema can MUST use its collection name as it scope also, so that one can query the specific schema belongs to one collection for convenience

ATTRIBUTE_MAP

ATTRIBUTE_MAP is a map of attribute. the `ATTRIBUTES` contains all possible use attribute data type. When create `Schema`, `Template` and `Collection`, the attribute should be passed in so that contracts knows how to serialize the data.

Assets

Assets are the core of the NFT standard. They reference a schema that is used to serialize the asset's data as well as a collection that they belong to.

They can also optionally reference a template, in which case the s erialized data of the template will be treated as if it also was part of the asset's data. Moreovder, if an asset use templete, both the template and the asset should follow the same schema to serialize data.

Scope of asset MUST use the owner of that asset as its scope, for which can easily query all NFT assets belong to specific account.

Finally, an asset properties are combine both data from template and data belongs to asset itself.

Issue&Burn Mechanism

Cause we use `Collection` to group same kinds of NFT asset, when creates a new kind of NFT asset, we can treat it as create a new `Collection`, and define operators to mint assets. Before issuing a new kind of NFT asset, we

MUST first create a new `Schema` through `createschema` action, then create a new `Collection` through `createcol` action and follows the `Schema` we create before. After that, if you have idea to some assets in same type, you can also create a template through `createtempl` function. Be aware of that if you have template for NFT asset, you should increase and decrease the supply info when you mint or reitre.

Burning NFT asset is allowed, but can only be done by the owner of that nft asset, anyone excepet the owner have no right to burn the asset.

Log

Like ERC721 standard, logs is also require when ACTION happen, and its convenience for third party to integrate. There are two basis log interface that every NFT contract must implements, logmint, logtransfer, logburnasset, lognewtempl and logsetdata ACTIONm corresponding mint, transfer, burn asset, new implementation and data seting ACTION.

Implementation

[Atomic asset](https://github.com/pinknetworkx/atomicassets-contract) project create a greate implementation of that standard, including some great serialization&descrialization library

Security Consideration

Fake deposit

Due to the notification feature of EOS itself, contracts that want to receive NFT token will implement the on_nftreceived function so that it can act some costomer logic on that. But when doing this, the `code` value of the incoming notification should be take into consideration in case of some fake deposit.

Overflow

Because the totalSupply use unsign value as its type, overflow will happen when try to subtract from 0. So when retire the tokens, must ensure that the value of totalSupply is greater than 0

Copyright