


ERC: Non-fungible Token Standard #721

New issue

 Open

dete opened this issue on Sep 22 · 64 comments



dete commented on Sep 22 • edited ▼

Latest draft. For context, the original draft is included at the bottom of this post hidden in the "Original Draft" disclosure widget.

Preamble

EIP: <to be assigned>
Title: Non-fungible Token Standard
Author: Dieter Shirley <dete@axiomzen.co>
Type: Standard
Category: ERC
Status: Draft
Created: 2017-09-20

Simple Summary

A standard interface for non-fungible tokens.

Abstract

This standard allows for the implementation of a standard API for non-fungible tokens (henceforth referred to as "NFTs") within smart contracts. This standard provides basic functionality to track and transfer ownership of NFTs.

Motivation

A standard interface allows any NFTs on Ethereum to be handled by general-purpose applications. In particular, it will allow for NFTs to be tracked in standardized wallets and traded on exchanges.

Specification

ERC-20 Compatibility

name

```
function name() constant returns (string name)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns the name of the collection of NFTs managed by this contract. - e.g. "My Non-Fungibles" .

symbol

```
function symbol() constant returns (string symbol)
```

Assignees

No one assigned

Labels

None yet

Projects

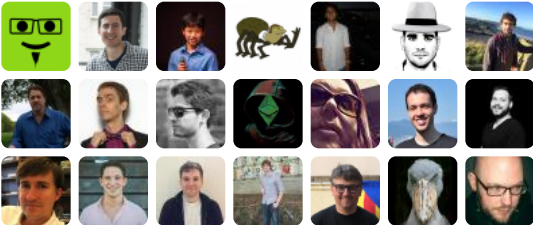
None yet

Milestone

No milestone

Notifications

29 participants



and others

*OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts **MUST NOT** depend on the existence of this method.*

Returns a short string symbol referencing the entire collection of NFTs managed in this contract. e.g. "MNFT". This symbol **SHOULD** be short (3-8 characters is recommended), with no whitespace characters or new-lines and **SHOULD** be limited to the uppercase latin alphabet (i.e. the 26 letters used in English).

totalSupply

```
function totalSupply() constant returns (uint256 totalSupply)
```

Returns the total number of NFTs currently tracked by this contract.

balanceOf

```
function balanceOf(address _owner) constant returns (uint256 balance)
```

Returns the number of NFTs assigned to address `_owner` .

Basic Ownership

ownerOf

```
function ownerOf(uint256 _tokenId) constant returns (address owner)
```

Returns the address currently marked as the owner of `_tokenId` . This method **MUST** `throw` if `_tokenId` does not represent an NFT currently tracked by this contract. This method **MUST NOT** return 0 (NFTs assigned to the zero address are considered destroyed, and queries about them should `throw`).

approve

```
function approve(address _to, uint256 _tokenId)
```

Grants approval for address `_to` to take possession of the NFT with ID `_tokenId` . This method **MUST** `throw` if `msg.sender != ownerOf(_tokenId)` , or if `_tokenId` does not represent an NFT currently tracked by this contract, or if `msg.sender == _to` .

Only one address can "have approval" at any given time; calling `approveTransfer` with a new address revokes approval for the previous address. Calling this method with 0 as the `_to` argument clears approval for any address.

Successful completion of this method **MUST** emit an `Approval` event (defined below) unless the caller is attempting to clear approval when there is no pending approval. In particular, an Approval event **MUST** be fired if the `_to` address is zero and there is some outstanding approval. Additionally, an Approval event **MUST** be fired if `_to` is already the currently approved address and this call otherwise has no effect. (i.e. An `approve()` call that "reaffirms" an existing approval **MUST** fire an event.)

Action	Prior State	<code>_to</code> address	New State	Event
Clear unset approval	Clear	0	Clear	None
Set new approval	Clear	X	Set to X	Approval(owner, X, tokenId)
Change approval	Set to X	Y	Set to Y	Approval(owner, Y, tokenId)
Reaffirm approval	Set to X	X	Set to X	Approval(owner, X, tokenId)

Clear approval	Set to X	0	Clear	Approval(owner, 0, tokenId)
----------------	----------	---	-------	-----------------------------

Note: ANY change of ownership of an NFT – whether directly through the `transfer` and `transferFrom` methods defined in this interface, or through any other mechanism defined in the conforming contract – MUST clear any and all approvals for the transferred NFT. The implicit clearing of approval via ownership transfer MUST also fire the event `Approval(0, _tokenId)` if there was an outstanding approval. (i.e. All actions that transfer ownership must emit the same Approval event, if any, as would emitted by calling `approve(0, _tokenId)` .)

takeOwnership

```
function takeOwnership(uint256 _tokenId)
```

Assigns the ownership of the NFT with ID `_tokenId` to `msg.sender` if and only if `msg.sender` currently has approval (via a previous call to `approveTransfer`). A successful transfer MUST fire the `Transfer` event (defined below).

This method MUST transfer ownership to `msg.sender` or `throw` , no other outcomes can be possible. Reasons for failure include (but are not limited to):

- `msg.sender` does not have approval for `_tokenId`
- `_tokenId` does not represent an NFT currently tracked by this contract
- `msg.sender` already has ownership of `_tokenId`

Important: Please refer to the Note in the `approveTransfer` method description; a successful transfer MUST clear pending approval.

transfer

```
function transfer(address _to, uint256 _tokenId)
```

Assigns the ownership of the NFT with ID `_tokenId` to `_to` if and only if `msg.sender == ownerOf(_tokenId)` . A successful transfer MUST fire the `Transfer` event (defined below).

This method MUST transfer ownership to `_to` or `throw` , no other outcomes can be possible. Reasons for failure include (but are not limited to):

- `msg.sender` is not the owner of `_tokenId`
- `_tokenId` does not represent an NFT currently tracked by this contract
- `_to` is 0 (Conforming contracts MAY have other methods to destroy or burn NFTs, which are conceptually "transfers to 0" and will emit `Transfer` events reflecting this. However, `transfer(0, tokenId)` MUST be treated as an error.)

A conforming contract MUST allow the current owner to "transfer" a token to themselves, as a way of affirming ownership in the event stream. (i.e. it is valid for `_to == ownerOf(_tokenId)` .) This "no-op transfer" MUST be considered a successful transfer, and therefore MUST fire a `Transfer` event (with the same address for `_from` and `_to`).

Important: Please refer to the Note in the `approveTransfer` method description; a successful transfer MUST clear pending approval. *This includes no-op transfers to the current owner!*

tokenOfOwnerByIndex

```
function tokenOfOwnerByIndex(address _owner, uint256 _index) constant returns (uint tokenId)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns the *n*th NFT assigned to the address `_owner` , with *n* specified by the `_index` argument. This method MUST `throw` if `_index >= balanceOf(_owner)` .

Recommended usage is as follows:

```
uint256 ownerBalance = nonFungibleContract.balanceOf(owner);

uint256[] memory ownerTokens = new uint256[](ownerBalance);

for (uint256 i = 0; i < ownerBalance; i++) {
    ownerTokens[i] = nonFungibleContract.tokenOfOwnerByIndex(owner, i);
}
```

Implementations MUST NOT assume that NFTs are accessed in any particular order by their callers (In particular, don't assume this method is called in a monotonically ascending loop.), and MUST ensure that calls to `tokenOfOwnerByIndex` are fully **idempotent** unless and until some non- `constant` function is called on this contract.

Callers of `tokenOfOwnerByIndex` MUST never assume that the order of NFTs is maintained outside of a single operation, or through the invocation (direct or indirect) of any non- `constant` contract method.

NOTE: Current limitations in Solidity mean that there is no efficient way to return a complete list of an address's NFTs with a single function call. Callers should not assume this method is implemented efficiently (from a gas standpoint) and should *strenuously avoid* calling this method "on-chain" (i.e. from any non- `constant` contract function, or from any `constant` contract function that is likely to be called on-chain).

NFT Metadata

tokenMetadata

```
function tokenMetadata(uint256 _tokenId) constant returns (string infoUrl)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns a **multiaddress** string referencing an external resource bundle that contains (optionally localized) metadata about the NFT associated with `_tokenId` . The string MUST be an IPFS or HTTP(S) base path (without a trailing slash) to which specific subpaths are obtained through concatenation. (IPFS is the preferred format due to better scalability, persistence, and immutability.)

Standard sub-paths:

- name (required) - The `name` sub-path MUST contain the UTF-8 encoded name of the specific NFT (i.e. distinct from the name of the collection, as returned by the contract's `name` method). A name SHOULD be 50 characters or less, and unique amongst all NFTs tracked by this contract. A name MAY contain white space characters, but MUST NOT include new-line or carriage-return characters. A name MAY include a numeric component to differentiate from similar NFTs in the same contract. For example: "Happy Token **#157**".
- image (optional) - If the `image` sub-path exists, it MUST contain a PNG, JPEG, or SVG image with at least 300 pixels of detail in each dimension. The image aspect ratio SHOULD be between 16:9 (landscape mode) and 2:3 (portrait mode). The image SHOULD be structured with a "safe zone" such that cropping the image to a maximal, central square doesn't remove any critical information. (The easiest way to meet this requirement is simply to use a 1:1 image aspect ratio.)
- description (optional) - If the `description` sub-path exists, it MUST contain a UTF-8 encoded textual description of the asset. This description MAY contain multiple lines and SHOULD use a single new-line character to delimit explicit line-breaks, and two new-line characters to delimit paragraphs. The description MAY include **CommonMark**-compatible Markdown annotations for styling. The description SHOULD be 1500 characters or less.
- other metadata (optional) - A contract MAY choose to include any number of additional subpaths, where they are deemed useful. There may be future formal and informal standards for additional

metadata fields independent of this standard.

Each metadata subpath (including subpaths not defined in this standard) MUST contain a sub-path `default` leading to a file containing the default (i.e. unlocalized) version of the data for that metadata element. For example, an NFT with the metadata path `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe` MUST contain the NFT's name as a UTF-8 encoded string available at the full path `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe/name/default` . Additionally, each metadata subpath MAY have one or more localizations at a subpath of an [ISO 639-1](#) language code (the same language codes used for HTML). For example, `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe/name/en` would have the name in English, and `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe/name/fr` would have the name in French (note that even localized values need to have a `default` entry). Consumers of NFT metadata SHOULD look for a localized value before falling back to the `default` value. Consumers MUST NOT assume that all metadata subpaths for a particular NFT are localized similarly. For example, it will be common for the `name` and `image` objects to not be localized even when the `description` is.

You can explore the metadata package referenced in this example [here](#).

Events

Transfer

This event MUST trigger when NFT ownership is transferred via any mechanism.

Additionally, the creation of new NFTs MUST trigger a Transfer event for each newly created NFTs, with a `_from` address of 0 and a `_to` address matching the owner of the new NFT (possibly the smart contract itself). The deletion (or burn) of any NFT MUST trigger a Transfer event with a `_to` address of 0 and a `_from` address of the owner of the NFT (now former owner!).

NOTE: A Transfer event with `_from == _to` is valid. See the `transfer()` documentation for details.

```
event Transfer(address indexed _from, address indexed _to, uint256 _tokenId)
```

Approval

This event MUST trigger on any successful call to `approve(address _spender, uint256 _value)` (unless the caller is attempting to clear approval when there is no pending approval).

See the documentation for the `approve()` method above for further detail.

```
event Approval(address indexed _owner, address indexed _approved, uint256 _tokenId)
```

Rationale

Utility

There are many proposed uses of Ethereum smart contracts that depend on tracking individual, non-fungible tokens (NFTs). Examples of existing or planned NFTs are LAND in [Decentraland](#), the eponymous punks in [CryptoPunks](#), and in-game items using systems like [Dmarket](#) or [EnjinCoin](#). Future uses include tracking real-world non-fungible assets, like real-estate (as envisioned by companies like [Ubitquity](#) or [Propy](#)). It is critical in each of these cases that these items are not "lumped together" as numbers in a ledger, but instead, each token must have its ownership individually and atomically tracked. Regardless of the nature of these items, the ecosystem will be stronger if we have a standardized interface that allows for cross-functional non-fungible token management and sales platforms.

NFT IDs

The basis of this standard is that every NFT is identified by a unique, 256-bit unsigned integer within its tracking contract. This ID number MUST NOT change for the life of the contract. The pair (contract address, asset ID) will then be a globally unique and fully-qualified identifier for a specific NFT within the Ethereum ecosystem. While some contracts may find it convenient to start with ID 0 and simply increment by one for each new NFT, callers MUST NOT assume that ID numbers have any specific pattern to them, and should treat the ID as a "black box".

Backwards Compatibility

This standard follows the semantics of ERC-20 as closely as possible, but can't be entirely compatible with it due to the fundamental differences between fungible and non-fungible tokens.

Example non-fungible implementations as of September 2017:

- [CryptoPunks](#) - Partially ERC-20 compatible, but not easily generalizable because it includes auction functionality directly in the contract and uses function names that explicitly refer to the NFTs as "punks".
- [Auctionhouse Asset Interface](#) - @dob needed a generic interface for his Auctionhouse dapp (currently ice-boxed). His "Asset" contract is very simple, but is missing ERC-20 compatibility, approve() functionality, and metadata. This effort is referenced in the discussion for [EIP-173](#).

(It should be noted that "limited edition, collectable tokens" like [Curio Cards](#) and [Rare Pepe](#) are *not* non-fungible tokens. They're actually a collection of individual fungible tokens, each of which is tracked by its own smart contract with its own total supply (which may be 1 in extreme cases).)

Implementation

Reference implementation forthcoming...

Copyright

Copyright and related rights waived via [CC0](#).

► Original Draft

	50		7
-------------------------------------------------------------------------------------	----	-------------------------------------------------------------------------------------	---



dob commented on Sep 22

You could consider using an [ipfs multiaddr](#) as the return value for the token metadata. This would allow for a self describing IPFS hash, http url, or swarm addr in the future.

	9
-------------------------------------------------------------------------------------	---



dete commented on Sep 22

Great point, @dob. Thanks for the suggestion! I've updated the draft above.

(For reference, my original posting had suggested using the format `ipfs:QmYwAP...` for IPFS links.)



kylerchin commented on Sep 23

wow! awesome integration! This is better than sending the raw data over the wire. :)



Arachnid commented on Sep 23

Collaborator

Nice proposal! For the metadata to be useful, though, I think you need to mandate its format, rather than just recommend it.

👍 7



GNSPS commented on Sep 23

A much needed EIP! 🙌

🎉 1



dip239 commented on Sep 23

I would propose to use term "asset" instead of "nun-fungible token" to separate both terms. It is an usual mistake to take some (fungible) token as representation of some (non-fungible) physical asset. So we can make it clear from very beginning.

👍 2



dete commented on Sep 23

@Arachnid: The whole metadata method is optional, but if someone is going to implement it, they should probably implement it in a way that everyone else is expecting. I anticipate using "SHOULD" language for the full specification.

@dip239: Ah naming things. Every programmer's favourite bikeshedding vortex. 😊 I bounced between three separate ideas for naming: "token" as seen above, "asset", following @dob's lead (and your suggestion!), and "NFT" which I used in the prose, but which is incredibly awkward in the API calls: `nftMetadata(uint _nftId)` .

"Asset" does seem like a decent choice, and my first personal draft used it. I discarded it for a two key reasons:

- The definition of "asset" in a financial context says nothing about fungibility. Cash is an asset (fungible), so are shares (fungible), so is real-estate (non-fungible). ˘_ (ヾ) _˘
- On the other hand, the definition of "asset" in a financial context *directly implies* financial value, something that must be declared on one's balance sheet. I believe one of the motivations between using the term "token" for ERC-20 instead of "coin" was specifically to avoid the implication of financial value at a time (which I would argue we are still in) when the crypto community wanted to avoid giving ammunition to traditional regulators—who don't really understand all the implications of what we're doing here.

I did spend some time trying out other alternatives ("item", "object", "thing", etc.), but none seemed right. I am happy to hear other suggestions because I am not entirely content with "token". It just seems like the best of some bad options.

👍 4



silasdavis commented on Sep 24

@dete your justification for NFT sounds rational to me, but allowing myself to plunge into the vortex for a moment. Sometimes finding a word that is relatively unreserved in modern usage but means the same thing works. 'Tesserae' were ancient tokens or tiles used as theatre tickets, forms of religious authentication, etc (see: https://www2.warwick.ac.uk/fac/arts/classics/research/dept_projects/tcam/about/, https://www2.warwick.ac.uk/fac/arts/classics/research/dept_projects/tcam/blog/, [https://en.wikipedia.org/wiki/Tessera_\(commerce\)](https://en.wikipedia.org/wiki/Tessera_(commerce))). Seems like they did function as a kind of non-fungible potentially value-holding token, but were quite varied in their specific application, which might suit the present case. Then again maybe it is just trying a bit too hard...



Arachnid commented on Sep 25 Collaborator

@dete I think you SHOULD use MUST instead. Otherwise, callers have no way to know how to interpret the return value correctly.

2

1



tjayrush commented on Sep 25 • edited ▼

Possible (but unlikely) names: ticket, badge, wafer, tile, marker



dete commented on Sep 28

@silasdavis: If this were a whole project, and not a single interface that is part of something larger, I'd jump on Tessera in a heartbeat. In that situation, a bit of an unusual name – with some history and context behind it – is pretty compelling. For something like an interface, tho, it's probably much better to stick with a term that people are already familiar with; even if it's imperfect.

@tjayrush: Thanks for the suggestions! I don't think most of them work well, but "marker" might. I'm definitely going to stew on that one a bit more. If it weren't overloaded in such common usage as a writing implement, it would probably be just about perfect.



dete commented on Sep 28

@Arachnid: Happy to follow the community lead here, but if you look at ERC-20, they use "SHOULD" language for something as foundational as emitting `Transfer` events. The metadata structure felt no more critical to me than `Transfer` ...

3



MicahZoltu commented on Sep 28 Contributor

Do *not* use ERC20 as an example of a good standard. A ton of people went and implemented the *draft* and then we had to finalize a "standard" that basically just listed what other people were doing. I don't believe anyone sees ERC20 as a *good* standard and everyone I have spoken to would like to see new better standards (hence why ERC223 exists).

Specifically, ERC20 used SHOULD because the standard came after a bunch of implementations and not all implementations did the same thing, so the standard couldn't say MUST without causing a bunch of ERC20-like tokens to not be ERC20.

9

4

dete commented on Sep 29



Great context, @MicahZoltu. Thank you!



Arachnid commented on Oct 2

Collaborator

@dete I'd also argue that the format of something is more foundational than whether you emit it or not. If you don't emit a transfer event, others can't track token transfers for your token - but if you don't require the format of a field, then they can't parse it *anywhere*.

👍 1



eordano commented on Oct 6 • edited ▼

Hey @dete! I went ahead and coded a first draft of this. Names are slightly different.
<https://github.com/decentraland/land/blob/master/contracts/BasicNFT.sol>

👍 4 🎉 3



dete commented on Oct 6

So, I was talking to @flockonus (another Solidity engineer here) and I was worrying about the "lost token" problem that ERC-223 tries to solve. (Essentially, ERC-20 has no mechanism to keep users from sending their tokens to contracts that don't know how to handle them, resulting in entirely unrecoverable coins. @Dexaran [did some analysis](#) estimating that something like \$400k has been lost in this way!)

The solution proposed by ERC-223 is to only allow transfers to contract addresses that implement the `tokenFallback` method. This has the nice side effect that the receiving contract gets a chance to do some work when it receives the coins.

I talked to @flockonus because I was hesitant to add more complexity to NFTs, and he kind of floored me with a suggestion that seemed crazy at first, but is *really* growing on me: Just get rid of the `transfer` method.

So hear me out: What if the only way to transfer an NFT was for the owner to `approve` a recipient, and have the recipient (whether contract or non-contract) call `takeOwnership` (which works like `transferFrom` , but with `msg.sender` hard-coded as the `_to` parameter).

This solves three problems, two of which are shared with fungible tokens (and addressed by ERC-223), one that is specific to NFTs:

1. Sending assets to contracts that don't know how to handle them. That becomes impossible if there is no direct `transfer` call; the contract *can't* be given a token, it has to call `takeOwnership` .
2. Contracts should know when they get new tokens. Well, since the contract has to explicitly `takeOwnership` , the flow for sending a token to a contract would be `approve(contract)` followed by `contract.someMethod()` . `someMethod` that would `takeOwnership` , plus the kind of things you'd put into `tokenFallback` (if appropriate).
3. Some low-value NFTs could end up being something akin to "spam". The easiest motivating example would be "trash" items in a video game; no game I know of allows griefers to shove items in your backpack without your consent. (This could be especially nasty in a smart-contract context where certain operations may be O(n) on the number of NFTs you hold.)

It introduces a new problem, of course: The simplest conceivable operation (transferring ownership from user to user) now requires two transactions instead of one. But how much of a problem is this, really? As time goes on, we'll see more and more transactions mediated by exchanges, smart contracts, UIs, and automated agents.

One possible option would be to include `transfer` but make it optional (meaning that contracts would know not to use it), and strongly recommend that it `throws` if targeting a contract address. (Although that wouldn't solve the spammy NFT problem.)

I'd love to hear other folks thoughts!

👍 5



Dexaran commented on Oct 7

@dete It's an interesting idea, but I think that the problems of this approach outweigh its advantages because:

1. Ethereum suffers bandwidth issues. I think that requiring that each token move is performed with two transactions is an irrational use of the blockchain.
2. A couple of transactions will require additional gas so it will be more expensive.
3. Approving then withdrawing is not a common pattern of sending funds in the cryptocurrency world. You can send ETH, ETC, Bitcoins, PIVX and even Doge without any confirmation from the side of the receiver. This will not be an intuitive-clear for users.
4. The ideology of uniformity. ETH and tokens are currencies. I think that it is better to make them behave similar.
5. The problems that you try to solve with this proposal are already solved with ERC223. I don't see any value in solving already-solved problems.

😐 1



ryanschneider commented on Oct 13

Some low-value NFTs could end up being something akin to "spam". The easiest motivating example would be "trash" items in a video game; no game I know of allows griefers to shove items in your backpack without your consent. (This could be especially nasty in a smart-contract context where certain operations may be $O(n)$ on the number of NFTs you hold.)

I thought about this some, and is it really that big of an issue? With the current ERC, each spam item would require its own transaction, so it would be a rather expensive attack, right?

That said, what if there were two new optional methods:

```
approveMultiple(address _to, uint[] _tokenIds)
transferMultipleFrom(address _from, address _to, uint[] _tokenIds)
```

These would allow "high volume" NFT contracts (like your MMO item example) to do "approved" bulk transfers, while the lower volume contracts would be naturally protected from spam via the requirement to send a single tokenId per transaction using the standard `transfer` call.

These methods would also allow one to merge/move wallets w/ 2 transactions instead of the N transactions currently required to move one item at a time.



flockonus commented on Oct 18 • edited ▼

About the point of removing the `transfer` method, we've been thinking heavily about this for CryptoKitties (disclosure i'm in Dete's team), and altho we have unconditional transfer method implemented for the alpha, it's easy to see why unconditional transfer might not work in multiple cases.

To start off, we have to understand that while a similar API to ERC20 is desirable, the case it aims to solve is different. In ERC20 the abstraction is *the more you have the better* because the tokens have positive monetary value, and go into an indistinguishable pile.

That's not the case with non-fungible, either with a smart contract that tracks property ownership such as the one [Dubai is implementing](#), or a game with relatively lower value assets, they might not always carry positive value or be desirable.

So even with due consideration about it being an expensive attack, when designing a mechanism we should take into account that some users will have significant more purchase power than others, and still they shouldn't be able to harm others with less.

My point is, if `transfer` is implemented it should be expected to throw depending on the smart contract implementation of what user decides to accept what. The business rules of accepting a token would definitely vary for each S.C. implementation.

 3



dete commented on Nov 8

Upon further reflection and discussion with [@flockonus](#), we are proposing to keep a straightforward (and therefore "unsafe") `transfer()` method. (i.e. It would work like ERC-20, and not [EIP-223](#).)

Our reasoning is as follows:

- Simpler is always better for standards. The fewer requirements, the harder it is to screw it up. (And NFTs are already complicated enough!)
- It is not the job of a smart contract to protect against every possible user error (it must protect against *invalid* actions, not *unintended* actions). In particular, the case we are trying to avoid (sending NFTs to contracts that don't know how to handle them) is better served by checks and warnings *in the wallet software*. A wallet (or other smart contract interface) will always be able to have more robust and dynamic checks than a smart contract, and is also able to engage in some "back-and-forth" dialog with the user. Smart contracts don't have the equivalent of a "This seems unsafe, are you sure?" dialog box!

Similarly, as [@flockonus](#) mentioned about the potential "spam" problem: This is not a problem that the community standard needs to solve. If an implementor of an ERC-721 contract feels like spam is likely to be a problem with their NFT, they are welcome to include additional functionality to make it easy for users to mark their accounts as not accepting unsolicited transfers.



ryanschneider commented on Nov 8

Is `transfer()` still accepting a single NFT tokenId? Any thoughts on my suggestion of defining some *optional* methods to support bulk transfers?

Say I decide to sell a large portion of my NFT collection to someone else, and it contains hundreds of NFTs (or more). Shouldn't there be a standard way to transfer `N` items atomically?

I can see the point of trying to keep the ERC simple, but am concerned that w/o a solution for bulk transfers the scope of what NFTs the ERC can be viable for is limited.



dete commented on Nov 9

Here is the new draft, which is basically the first "complete" draft. Any and all comments are welcome!

Preamble

EIP: <to be assigned>
Title: Non-fungible Token Standard
Author: Dieter Shirley <dete@axiomzen.co>
Type: Standard
Category: ERC
Status: Draft
Created: 2017-09-20

Simple Summary

A standard interface for non-fungible tokens.

Abstract

This standard allows for the implementation of a standard API for non-fungible tokens (henceforth referred to as "NFTs") within smart contracts. This standard provides basic functionality to track and transfer ownership of NFTs.

Motivation

A standard interface allows any NFTs on Ethereum to be handled by general-purpose applications. In particular, it will allow for NFTs to be tracked in standardized wallets and traded on exchanges.

Specification

ERC-20 Compatibility

name

```
function name() constant returns (string name)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns the name of the collection of NFTs managed by this contract. - e.g. "My Non-Fungibles" .

symbol

```
function symbol() constant returns (string symbol)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns a short string symbol referencing the entire collection of NFTs managed in this contract. e.g. "MNFT". This symbol SHOULD be short (3-8 characters is recommended), with no whitespace characters or new-lines and SHOULD be limited to the uppercase latin alphabet (i.e. the 26 letters used in English).

totalSupply

```
function totalSupply() constant returns (uint256 totalSupply)
```

Returns the total number of NFTs currently tracked by this contract.

balanceOf

```
function balanceOf(address _owner) constant returns (uint256 balance)
```


Returns the number of NFTs assigned to address `_owner` .

Basic Ownership

ownerOf

```
function ownerOf(uint256 _tokenId) constant returns (address owner)
```

Returns the address currently marked as the owner of `_tokenId` . This method MUST `throw` if `_tokenId` does not represent an NFT currently tracked by this contract. This method MUST NOT return 0 (NFTs assigned to the zero address are considered destroyed, and queries about them should `throw`).

approve

```
function approve(address _to, uint256 _tokenId)
```

Grants approval for address `_to` to take possession of the NFT with ID `_tokenId` . This method MUST `throw` if `msg.sender != ownerOf(_tokenId)` , or if `_tokenId` does not represent an NFT currently tracked by this contract, or if `msg.sender == _to` .

Only one address can "have approval" at any given time; calling `approveTransfer` with a new address revokes approval for the previous address. Calling this method with 0 as the `_to` argument clears approval for any address.

Successful completion of this method MUST emit an `Approval` event (defined below) unless the caller is attempting to clear approval when there is no pending approval. In particular, an Approval event MUST be fired if the `_to` address is zero and there is some outstanding approval. Additionally, an Approval event MUST be fired if `_to` is already the currently approved address and this call otherwise has no effect. (i.e. An `approve()` call that "reaffirms" an existing approval MUST fire an event.)

Action	Prior State	<code>_to</code> address	New State	Event
Clear unset approval	Clear	0	Clear	None
Set new approval	Clear	X	Set to X	Approval(owner, X, tokenId)
Change approval	Set to X	Y	Set to Y	Approval(owner, Y, tokenId)
Reaffirm approval	Set to X	X	Set to X	Approval(owner, X, tokenId)
Clear approval	Set to X	0	Clear	Approval(owner, 0, tokenId)

Note: ANY change of ownership of an NFT – whether directly through the `transfer` and `transferFrom` methods defined in this interface, or through any other mechanism defined in the conforming contract – MUST clear any and all approvals for the transferred NFT. The implicit clearing of approval via ownership transfer MUST also fire the event `Approval(0, _tokenId)` if there was an outstanding approval. (i.e. All actions that transfer ownership must emit the same Approval event, if any, as would emitted by calling `approve(0, _tokenId)` .)

takeOwnership

```
function takeOwnership(uint256 _tokenId)
```

Assigns the ownership of the NFT with ID `_tokenId` to `msg.sender` if and only if `msg.sender` currently has approval (via a previous call to `approveTransfer`). A successful transfer MUST fire the `Transfer` event (defined below).

This method MUST transfer ownership to `msg.sender` or `throw`, no other outcomes can be possible. Reasons for failure include (but are not limited to):

- `msg.sender` does not have approval for `_tokenId`
- `_tokenId` does not represent an NFT currently tracked by this contract
- `msg.sender` already has ownership of `_tokenId`

Important: Please refer to the Note in the `approveTransfer` method description; a successful transfer MUST clear pending approval.

transfer

```
function transfer(address _to, uint256 _tokenId)
```

Assigns the ownership of the NFT with ID `_tokenId` to `_to` if and only if `msg.sender == ownerOf(_tokenId)`. A successful transfer MUST fire the `Transfer` event (defined below).

This method MUST transfer ownership to `_to` or `throw`, no other outcomes can be possible. Reasons for failure include (but are not limited to):

- `msg.sender` is not the owner of `_tokenId`
- `_tokenId` does not represent an NFT currently tracked by this contract
- `_to` is 0 (Conforming contracts MAY have other methods to destroy or burn NFTs, which are conceptually "transfers to 0" and will emit `Transfer` events reflecting this. However, `transfer(0, tokenId)` MUST be treated as an error.)

A conforming contract MUST allow the current owner to "transfer" a token to themselves, as a way of affirming ownership in the event stream. (i.e. it is valid for `_to == ownerOf(_tokenId)`.) This "no-op transfer" MUST be considered a successful transfer, and therefore MUST fire a `Transfer` event (with the same address for `_from` and `_to`).

Important: Please refer to the Note in the `approveTransfer` method description; a successful transfer MUST clear pending approval. *This includes no-op transfers to the current owner!*

tokenOfOwnerByIndex

```
function tokenOfOwnerByIndex(address _owner, uint256 _index) constant returns (uint tokenId)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns the *n*th NFT assigned to the address `_owner`, with *n* specified by the `_index` argument. This method MUST `throw` if `_index >= balanceOf(_owner)`.

Recommended usage is as follows:

```
uint256 ownerBalance = nonFungibleContract.balanceOf(owner);

uint256[] memory ownerTokens = new uint256[](ownerBalance);

for (uint256 i = 0; i < ownerBalance; i++) {
    ownerTokens[i] = nonFungibleContract.tokenOfOwnerByIndex(owner, i);
}
```

Implementations MUST NOT assume that NFTs are accessed in any particular order by their callers (In particular, don't assume this method is called in a monotonically ascending loop.), and MUST ensure that calls to `tokenOfOwnerByIndex` are fully **idempotent** unless and until some non-`constant` function is called on this contract.

Callers of `tokenOfOwnerByIndex` MUST never assume that the order of NFTs is maintained outside of a single operation, or through the invocation (direct or indirect) of any non-`constant` contract method.

NOTE: Current limitations in Solidity mean that there is no efficient way to return a complete list of an address's NFTs with a single function call. Callers should not assume this method is implemented efficiently (from a gas standpoint) and should *strenuously avoid* calling this method "on-chain" (i.e. from any non-`constant` contract function, or from any `constant` contract function that is likely to be called on-chain).

NFT Metadata

tokenMetadata

```
function tokenMetadata(uint256 _tokenId) constant returns (string infoUrl)
```

OPTIONAL - It is recommend that this method is implemented for enhanced usability with wallets and exchanges, but interfaces and other contracts MUST NOT depend on the existence of this method.

Returns a `multiaddress` string referencing an external resource bundle that contains (optionally localized) metadata about the NFT associated with `_tokenId` . The string MUST be an IPFS or HTTP(S) base path (without a trailing slash) to which specific subpaths are obtained through concatenation. (IPFS is the preferred format due to better scalability, persistence, and immutability.)

Standard sub-paths:

- name (required) - The `name` sub-path MUST contain the UTF-8 encoded name of the specific NFT (i.e. distinct from the name of the collection, as returned by the contract's `name` method). A name SHOULD be 50 characters or less, and unique amongst all NFTs tracked by this contract. A name MAY contain white space characters, but MUST NOT include new-line or carriage-return characters. A name MAY include a numeric component to differentiate from similar NFTs in the same contract. For example: "Happy Token `#157`".
- image (optional) - If the `image` sub-path exists, it MUST contain a PNG, JPEG, or SVG image with at least 300 pixels of detail in each dimension. The image aspect ratio SHOULD be between 16:9 (landscape mode) and 2:3 (portrait mode). The image SHOULD be structured with a "safe zone" such that cropping the image to a maximal, central square doesn't remove any critical information. (The easiest way to meet this requirement is simply to use a 1:1 image aspect ratio.)
- description (optional) - If the `description` sub-path exists, it MUST contain a UTF-8 encoded textual description of the asset. This description MAY contain multiple lines and SHOULD use a single new-line character to delimit explicit line-breaks, and two new-line characters to delimit paragraphs. The description MAY include `CommonMark`-compatible Markdown annotations for styling. The description SHOULD be 1500 characters or less.
- other metadata (optional) - A contract MAY choose to include any number of additional subpaths, where they are deemed useful. There may be future formal and informal standards for additional metadata fields independent of this standard.

Each metadata subpath (including subpaths not defined in this standard) MUST contain a sub-path `default` leading to a file containing the default (i.e. unlocalized) version of the data for that metadata element. For example, an NFT with the metadata path `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe` MUST contain the NFT's name as a UTF-8 encoded string available at the full path `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe/name/default` . Additionally, each metadata subpath MAY have one or more localizations at a subpath of an [ISO 639-1](#) language code (the same language codes used for HTML). For example, `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe/name/en` would have the name in English, and `/ipfs/QmZU8bKEG8fhcQwKoLHfjtJoKBzvUT5LFR3f8dEz86WdVe/name/fr` would have the name in French (note that even localized values need to have a `default` entry). Consumers of NFT metadata SHOULD look for a localized value before falling back to the `default` value. Consumers MUST NOT assume that all metadata subpaths for a particular NFT are localized similarly. For example, it will be common for the `name` and `image` objects to not be localized even when the `description` is.

You can explore the metadata package referenced in this example [here](#).

Events

Transfer

This event MUST trigger when NFT ownership is transferred via any mechanism.

Additionally, the creation of new NFTs MUST trigger a Transfer event for each newly created NFTs, with a `_from` address of 0 and a `_to` address matching the owner of the new NFT (possibly the smart contract itself). The deletion (or burn) of any NFT MUST trigger a Transfer event with a `_to` address of 0 and a `_from` address of the owner of the NFT (now former owner!).

NOTE: A Transfer event with `_from == _to` is valid. See the `transfer()` documentation for details.

```
event Transfer(address indexed _from, address indexed _to, uint256 _tokenId)
```

Approval

This event MUST trigger on any successful call to `approve(address _spender, uint256 _value)` (unless the caller is attempting to clear approval when there is no pending approval).

See the documentation for the `approve()` method above for further detail.

```
event Approval(address indexed _owner, address indexed _approved, uint256 _tokenId)
```

Rationale

Utility

There are many proposed uses of Ethereum smart contracts that depend on tracking individual, non-fungible tokens (NFTs). Examples of existing or planned NFTs are LAND in [Decentraland](#), the eponymous punks in [CryptoPunks](#), and in-game items using systems like [Dmarket](#) or [EnjinCoin](#). Future uses include tracking real-world non-fungible assets, like real-estate (as envisioned by companies like [Ubitquity](#) or [Propy](#)). It is critical in each of these cases that these items are not "lumped together" as numbers in a ledger, but instead, each token must have its ownership individually and atomically tracked. Regardless of the nature of these items, the ecosystem will be stronger if we have a standardized interface that allows for cross-functional non-fungible token management and sales platforms.

NFT IDs

The basis of this standard is that every NFT is identified by a unique, 256-bit unsigned integer within its tracking contract. This ID number MUST NOT change for the life of the contract. The pair `(contract address, asset ID)` will then be a globally unique and fully-qualified identifier for a specific NFT within the Ethereum ecosystem. While some contracts may find it convenient to start with ID 0 and simply increment by one for each new NFT, callers MUST NOT assume that ID numbers have any specific pattern to them, and should treat the ID as a "black box".

Backwards Compatibility

This standard follows the semantics of ERC-20 as closely as possible, but can't be entirely compatible with it due to the fundamental differences between fungible and non-fungible tokens.

Example non-fungible implementations as of September 2017:

- [CryptoPunks](#) - Partially ERC-20 compatible, but not easily generalizable because it includes auction functionality directly in the contract and uses function names that explicitly refer to the NFTs as "punks".
- [Auctionhouse Asset Interface](#) - [@dob](#) needed a generic interface for his Auctionhouse dapp (currently ice-boxed). His "Asset" contract is very simple, but is missing ERC-20 compatibility,

approve() functionality, and metadata. This effort is referenced in the discussion for [EIP-173](#).

(It should be noted that "limited edition, collectable tokens" like [Curio Cards](#) and [Rare Pepe](#) are *not* non-fungible tokens. They're actually a collection of individual fungible tokens, each of which is tracked by its own smart contract with its own total supply (which may be `1` in extreme cases).)

Implementation

Reference implementation forthcoming...

Copyright

Copyright and related rights waived via [CC0](#).

 24



Arachnid commented on Nov 9

Collaborator

@dete You should update the initial issue with this draft, so people coming new to it don't need to scroll through all the comments.

 8



dete commented on Nov 9

Thanks @Arachnid: I put a link in the first comment for now, I don't want to lose the older version for anyone who wants to follow the conversation.



dete commented on Nov 9

Does anyone have any thoughts on whether the `_tokenId` in the events should be indexed? I would think yes (it's easy to imagine that people or processes would want to watch for events relating to specific NFTs), but this comment in the [Solidity docs on Events](#) gave me pause:

NOTE: Indexed arguments will not be stored themselves. You can only search for the values, but it is impossible to retrieve the values themselves.

That seems... problematic...



Arachnid commented on Nov 10

Collaborator

I don't believe that's quite accurate; you should be able to retrieve the contents of an indexed event (for a fixed length type) or its hash (for string/bytes/arrays).

 1



NoahMarconi commented on Nov 10

Any thoughts on transferring multiple tokens in a single transfer?

We ran into this needs recently and the gas costs are making many options unfeasible.



jpitts commented on Nov 10

Owner

For metadata, I would strongly recommend requiring a standard URI.

Perhaps consider a way to allow for linkage to the same metadata hosted on different storage networks (i.e. additional parameter specifying 'ipfs', 'swarm', or 'https'). This way the metadata has a better chance of persisting.

Additionally, instead of an external resource bundle, I would recommend using JSON-LD. This format allows for more complex data, has wide use and tooling (being JSON), and provides context to that data

<https://json-ld.org/>.

👍 3 👎 1



nadavhollander commented on Nov 10 • edited ▼

First -- greed with @jpitts on creating redundant storage of metadata on different storage networks and, more importantly IMO, being more unopinionated with respect to the data storage network. I imagine a URI format along the lines of "swarm://XYZ" / "ipfs://XYZ" with the URI prefix specifying the specific storage network queried.

Secondly, I'm working on tokenized debt issuance protocol (Dharma) and a big feature that would be extremely helpful to us is having built-in optional fungibility *within* each non-fungible asset. This may sound at odds with the purpose of NFTs, but I'll use Dharma as a tangible example:

Alice wants to issue a bond token using Dharma, and she wants to be able to sell individual shares in that bond to different creditors. She *could* ostensibly have an NFT representing the debt asset as a whole, wrap that NFT into another fungible ERC20 token contract, and sell tokens from that contract to creditors, but that would require her to incur the gas associated with deploying an entirely new token contract for a set of very generic token transfer functionality. It would be much simpler and cheaper to be able to ask the contract to mint an NFT with its own fractional supply of X tokens. In a sense, this would mean that individual debts would be non-fungible with one another, but *within* each debt there would be a fractional supply of tokens that *are* fungible with one another.

As an example, the function interface for `transfer` would look like this:

```
function transfer(address _to, uint256 _tokenId, uint246 _amount)
```

The standard could easily be made compatible with NFTs that have no fractional supply -- their issuance would simply have a fractional supply of 1. This is an advantageous arrangement for classes of digital assets that have (1) highly generic functionality and (2) require some sort of fractional fungibility and (2) don't merit the deployment of a smart contract for each issuance event.

With all the above being said, I would definitely not say this is a **must have** -- this arguably extends beyond the definition of non-fungibility to a certain degree. **If, however, this is a very common need among projects that have similar dynamics to the tokens they're issuing, I think this would be the appropriate standard in which to include this functionality.**

❤️ 5



dete commented on Nov 10

@NoahMarconi: I'd be curious if you could share more details about your scenario. Needless to say, specific NFTs can implement any additional functionality that meets their requirements, including bulk transfers. The question is: What is the value of putting this into shared spec? I'm currently not imagining a common scenario where someone would want to bulk transfer generic NFTs.

nadavhollander commented on Nov 10



Also -- thank you @dete for spearheading this 👍



1



2



willwarren89 commented on Nov 11

Hi @dete, great job putting together this spec.

I'm in favor of preserving `transferFrom(...)` and removing `takeOwnership(...)` for a few reasons:

1. I believe there are/will be numerous use cases where `transferFrom(...)` is necessary. For example, adding support for NFT tokens in 0x protocol is much easier with `transferFrom(...)` present.
2. Including `transferFrom(...)` increases the similarity between ERC721 and ERC20 token interfaces.
3. `takeOwnership(...)` is a special case of `transferFrom(...)`.



11



dete commented on Nov 11

consider a way to allow for linkage to the same metadata hosted on different storage networks

Well, my motivation is to put as little metadata as possible into the smart contract. Even just storing a IPFS pointer is likely to be pretty expensive (about two words worth), and having one of these for each NFT will add up pretty darned quickly.

On the other hand, I see the value. What do you think about something like this:

```
function tokenMetadata(uint256 _tokenId, string preferredTransport) constant returns (string infoUr
```

Where `preferredTransport` is something like "http", "ipfs", or "swarm". The contract doesn't NEED to return a URI matching `preferredTransport`, but it should try if it can. (If the contract doesn't know about the preferred transport requested by the caller, it just returns whatever transport it wants.)

instead of an external resource bundle, I would recommend using JSON-LD

Hrm. I'll be honest, that seems to add a lot of complexity, both for any service emitting the metadata, as well as for consumers wanting to ingest it.

I'm trying to hold the standard to least common denominator. I can well imagine that some implementors will see good reason to use something like JSON-LD, but is that really the best choice for everyone? I tried to pick a metadata specification that would make sense for most NFT creators and would still be easy, robust, and efficient for wallet and exchange providers. Happy to hear further thoughts on this...



eordano commented on Nov 13

One thing to note: For wallets, using the `balanceOf` and `tokenOfOwnerByIndex` function might lead to race conditions where the state of owned tokens mutates while the wallet was fetching the contents. Consider adding a new function, to retrieve the full list of tokenIds, just for the purposes of queries that come from applications and not the EVM.



NoahMarconi commented on Nov 13

@dete The scenario I'm looking at is IoT related tying tokens to tangible assets. The assets are low enough value that ownership transfers are likely to happen almost exclusively in bulk. Conversely, the assets are high enough value that they warrant tracking ownership provenance.

Regardless of which spec we follow, there's still the cost to update state on the blockchain for each transfer of ownership. Given the volume of state changes required we're leaning toward side chains, perhaps, with a blockchain that natively supports NFTs.



dete commented on Nov 13

Believe it or not, I have given some thought to tokenization of NFTs! Tokenization probably doesn't matter much for [our project](#) 🐱 but it's definitely something that comes to mind when tracking tangible assets.

My primary argument against including this capability in ERC-721 is simple: [KISS](#). 😬 I feel like the goal for a standard like this is to collect the functionality shared across *the overwhelming majority* of potential implementers. While tokenization is hardly unthinkable, I also believe that there are many interesting NFT use-cases that don't benefit from tokenization. It's also worth pointing out that tokens issued in this way would distinctly *not* be ERC-20 compatible, and could never be.

What I would like to see, however, is a public TokenizeNonFungible contract, ERC-20 compatible, that can be instantiated for any specific NFT. It could take the NFT contract address and an NFT ID, and then track shares of that NFT in the typical ERC-20 fashion. I feel like this contract could be fairly simple and relatively cheap to deploy, especially if you were able to factor out some of the code into a library contract (if I dare mention that idea so soon after the most recent Parity disaster... 😬).

👍 1



dete commented on Nov 13

@NoahMarconi Makes sense. I would encourage you to create a (probably informal) derivative standard that extends ERC-721 with bulk transfers. Remember, ERC's don't restrict you from having domain-specific functionality!

I still maintain that *requiring* that all ERC-721 tokens support bulk transfers isn't ideal, but I agree that if it's a likely scenario, there could be some pretty significant gas savings.



dete commented on Nov 13

BTW - In case my 👍s weren't obvious, the next draft will use `transferFrom` (instead of `takeOwnership`) and standard URLs (instead of multiaddresses).

👍 5



dete commented on Nov 13

@eordano:

Consider adding a new function, to retrieve the full list of tokenIds, just for the purposes of queries that come from applications and not the EVM.

Yeah, as we've seen from implementing our own token, I think we'll need to make this change.

Storing a list of NFTs by owner address is pretty expensive (for storage) and a pain to maintain, walking through the complete list of NFTs checking ownership is *also* expensive (execution cost, this time) and [DoS prone](#). I wanted to avoid returning an array because that [doesn't work on-chain](#), but it turns out that any "normal" implementation of `tokenOfOwnerByIndex` can't be called efficiently on-chain anyway (and again, risks DoS).

I think it's best to replace `tokenOfOwnerByIndex` with `getTokensOfOwner` that returns a dynamic array, and simply state that this MUST never be called by a smart contract. (And even if anyone tries it, they won't get anything useful out of it...)



willwarren89 commented on Nov 13 • edited ▼

One potential issue that hasn't been discussed yet: the current ERC721 interface does not provide an easy way for other smart contracts to differentiate between an ERC20 and an ERC721. This makes it hard to write smart contracts that can handle both types of tokens.

It would be ideal if the ERC721 interface included a flag `isERC721`, but it would require some funky calling behavior to check for this flag on an ERC20 token without an exception being thrown. The following code would need to be used:

```
if(tokenAddress.call(bytes4(sha3("isERC721()")))) { ... };
```

If `tokenAddress` is an ERC721 token that includes the flag and sets it to true, the above should return `true`. If `tokenAddress` is an ERC20 token that does not contain this flag in its interface, the above should return `false` rather than throw an exception.

👍 7



dete commented on Nov 15

If anyone is interested in seeing the source code for our implementation of ERC-721, please check here: <https://github.com/axiomzen/cryptokitties-bounty/blob/master/contracts/KittyOwnership.sol>

👍 9



willwarren89 commented 29 days ago

Regarding the addition of a flag `isERC721` or `implementsERC721`: [@PhABC](#) has pointed out that a more general solution to this issue has already been proposed: [Pseudo-Introspection, or standard interface detection](#). I'm in favor of adopting this more general solution.

👍 4



abandeali1 commented 29 days ago

I think it's worth considering adding a `data` parameter to all of the transfer functions, similar to ERC [#223](#) (or at least overload the current transfer functions). Since we're creating a new standard from scratch, no reason to hold off on features.

I also want to stress the importance of having some sort of ERC721 identifier in the contract. NFTs act in a very similar way to ERC20, but they should be handled completely differently by other contracts. I can think of at least a few examples of dangerous behavior resulting from contracts assuming that NFTs act the same way as an ERC20/223 token. It might actually be desirable for the standard interface to *not* resemble ERC20 (although forcing every standard to have a different interface probably isn't sustainable).

The nice thing about adding a simple flag is that it is completely backwards compatible (ERC20 will throw when trying to access the nonexistent flag). Something like EIP [#165](#) would result in false negatives for existing tokens, since they don't already include support for pseudo introspection.



jbaylina commented 28 days ago

What about just use a standard token (ERC20, ERC223, or whatever) with a total supply of only 1 wei ?

If you want, then you just create a "Factory contract" to create assets that are mainly ProxyContracts with a simple delegate call to a library contract that just acts as template.

👍 1



 dete referenced this issue 28 days ago

Pseudo-Introspection, or standard interface detection #165

 Open



dete commented 21 days ago

Thanks for thinking about creative solutions for this, @jbaylina!

We just went live with [our implementation](#) of ERC-721, and our users have created more than 2k NFTs in one day. I feel like issuing a new contract for each of these items would have been incredibly expensive.

But even if I'm overestimating the cost of contract creation, the *functionality* of our NFTs would have been much more expensive if every interaction between two NFTs ("breeding" in our case) would have required multiple cross-contract calls.

Finally, the complexity of building common wallet and exchange software seems like it would be much trickier, also. Not only would these things need some mechanism for enumerating all contracts that are "supposed" to be part of a particular NFT collection, it would need a mechanism for rejecting contracts that were *not* issued properly. Solvable, certainly, but also complex and potentially expensive.

👍 4



eordano commented 19 days ago • edited ▼

In the spirit of standardization, I've turned Decentraland's Land contract compatible with these methods. Some notes:

- These are our interfaces for the generic implementation:
<https://github.com/decentraland/land/blob/master/contracts/NFT.sol>
- The generic implementation can be found here:
<https://github.com/decentraland/land/blob/master/contracts/BasicNFT.sol>
- Our particular implementation extends that contract and defines a couple extra methods, for ease of use with x, y coordinates:
<https://github.com/decentraland/land/blob/master/contracts/LANDToken.sol>
- We use an array of tokens owned by user, so `tokenOfOwnerByIndex` is efficient. Creating tokens takes ~120,000 gas.
- We added the `getAllTokens(address owner) public constant returns (uint[])` function for querying all of a users' tokens.
- Our `tokenMetadata(uint tokenId)` function doesn't take the `preferredMethod` parameter, in the spirit of KISS.

Hope this is helpful, looking forward to hearing your feedback @dete!

👍 7



eordano commented 19 days ago • edited ▼

Also, I think we should be wary of race conditions on calling `approve` , and block reorgs.

Sources:

[#20 \(comment\)](#)

<https://github.com/Giveth/minime/blob/master/contracts/MiniMeToken.sol#L253>



PhABC commented 18 days ago

@dete Do you mind updating the OP with the newest draft (published 18 days ago)? Would be easier for people to understand that that's the current specification.



1



izqui referenced this issue in [aragon/aragon-apps](#) 13 days ago

Add ERC721 support to Vault #44



Open



superphly commented 11 days ago • edited ▼

Not all NFTs are positive. What if, and stay with me here, an NFT had a negative effect. With tokens, we treat them as net positives. The logic of "who would be mad if someone deposited money into your bank account" rings true. But what if that NFT was something like a debt or in the case of a game a "negative spell". Another example might be "I don't want ownership of that piece of property, I can't afford the tax bill" or "I have a moral objection to owning a gun".

Sorry, this is a concern about not being able to refuse a NFT. Maybe make this a feature of the contract? Some can be claimable.



7



simondlr commented 9 days ago

@dete Awesome stuff!

Can I propose that we have the most up to date reference in the top post? The link at the top already links to an out of date reference (using `takeOwnership` vs `transferFrom`).



leotianlizhan referenced this issue in [ethers-io/ethers.js](#) 9 days ago

parsing event error #86



Open



onetom commented 9 days ago

@dete There are a few references to `approveTransfer` but there is only an `approve` method defined in the document. What am I missing?



onetom commented 9 days ago

@jpitts I'm not really in favour of `json-ld` , because there are more well-thought-out alternatives with also quite [wide range of supported languages](#), for example: <https://github.com/edn-format/edn>
Sticking to JSON, just anchors us stronger into this suboptimal place where JavaScript sucked us into. It's a bit like: just because humanity piled up a lot of plastic bottle waste, we shouldn't start building everything out of plastic bottles.



moodysaleem commented 8 days ago • edited ▼

We need a way to transfer or approve more than one token at a time in the spec.

- `transfer` and `approve` and `takeOwnership` methods should allow to be called with an array of token IDs

We also need a way to notify recipient contracts of an approval in a single transaction, similar to how it's implemented in ERC20

- `approveAndCall(address _to, uint[] _tokenIds, bytes _extraData)`
 - e.g. you could implement an `AuctionHouse` contract by encoding the `AuctionHouse` sale call into the `extraData` parameter



mgraczyk commented 8 days ago

I understand that this EIP addresses a real need for wallet software. However, if it were framed more like "this is a read-only key-value store with object-level permissions", I think it would be even more useful as a building block for other smart contracts.

Specifically, `tokenMetadata` is "looking up an object", and `transfer` is "modify the owner of a key". With less token-specific language, I think this EIP would describe a more composable interface. The types of "values" stored could be seen as a property of each data store instantiation, rather than an inherent aspect of the interface. Users may want to build a key-value store whose values are something other than an URL. For example, the data store might hold `tokenId` s for tokens in other data stores with the same interface, or documents which contain links to other documents.



1



3



ricmoo commented 7 days ago

I think the enumeration API is wildly useful to offer (optionally), for wallets and other user interfaces for the purpose of not maintaining a separate database to cache the keys, and to provider authoritative data. I think there needs to be two (both of which could be optional, but either is sufficient) operations to iterate.

The current (which, for performance, is assuming tokens are contained in an array):

```
function tokenOfOwnerByIndex(address _owner, uint256 _index) constant returns (uint tokenId)
```

And an additional options (which allows for linked-list elements):

```
function tokenOfOwnerAfterToken(address _owner, uint256 previous tokenId) constant returns (uint t
```

For example, I have a struct with plenty of room to spare, and have a hard cap of tokenId at uint48, which means for free (ish) I can store a next and previous pointer to maintain a doubly linked-list, so insertion and deletion are O(1) and iteration from one item to the next is O(1), but I would need to store additional data to allow access by index.

There may be some need to have an additional call to get the "first" token for an address, or have a canonical tokenId that represents the null token (either 0 or $2^{256} - 1$).



2



This was referenced 5 days ago

- AuctionHouse Improvement Proposal: Additional in protocol incentives

dob/auctionhouse#31

Open
- AuctionHouse Improvement Proposal: Standard for non-fungible, on-chain assets

dob/auctionhouse#30

Open
- Support ERC 721 Assets

dob/auctionhouse#39

Open



drandreaskrueger commented 3 days ago

We suggest

```
tokensOfOwnerByIndex(address _owner, uint _index) constant returns (bytes32 tokenHash)
```

instead, so any arbitrary ID type can be represented.



optimizer999 commented 2 days ago

It seems one implementation of NFTs would be the real world mapping of assets to the blockchain. It is likely real world assets will contain their own reference numbers or the real world characteristics of the asset will be hashed to create a unique value.

As a thought experiment, consider property deeds that have been recorded in a county and assigned a number (Perhaps Book and Page).

If tracked by a NFT they would be referenced and transfered by their government assigned number, not by a tokenId in a contract.

While easy to implement a hashToIndex mapping to track real world identifiers in NFTs, this creates another level of necessary interaction with the contract.

1. Get index from hash
2. Approve Transfer using index
3. Transfer index From address

Any thoughts on how best to incorporate the tracking of real world identifiers in NFTs?



nadavhollander referenced this issue in [dharmaprotocol/charta](#) 2 days ago

Add RepaymentRouter #1

Open



abandeali1 commented 2 days ago • edited ▼

I think the current implementation of `approve` is pretty lacking in functionality. Generally, token owners will want to grant approval to a smart contract to transfer *any* of their NFTs, not just a single one. It is highly inefficient and poor UX to require an `approve` call for each individual NFT.

There needs to be either an `approveAll` function with a separate mapping OR we can special case a certain tokenId that would represent an approval for all NFTs (i.e $2^{256} - 1$ represents an approval for all tokenIds).

1



moodysalem commented 2 days ago

@abandeali1 I don't think there's a good use case to approve/transfer `a11` of your tokens (same with ERC20), but I do think there are many use cases to approve N specific NFTs. Plus it would be hard to implement `approveA11` without creating a function that has unbounded gas usage, or creating a vulnerability by giving the approved address access to all newly acquired tokens.

I asked for the ability to approve N tokens in [#issuecomment-350157122](#).



abandeali1 commented 2 days ago • edited ▼

@moodysaalem , it would be easy to implement with bounded gas usage with a separate `allowedA11` mapping. When checking allowances, the contract would first check `allowedA11` and then fall back to `allowed` . There are other ways to do this (special casing a tokenId), as mentioned in my previous comment [#721 \(comment\)](#). This *would* give the approved address access to all newly acquired tokens, but the most common use case of `approve` involves granting an approval to smart contracts that can still only access your tokens with your permission.

I also strongly disagree that there aren't many use cases for approving all of your tokens at once. Take a simple escrow contract, for example. Typically, you would call `approve` once, and then can deposit tokens into the contract any number of times by calling `deposit` . With the current implementation, you would have to call `approve` and `deposit` every single time you wanted to deposit an NFT into the escrow contract.

A more concrete example involves how we actually intend on using NFTs in 0x. In the current ERC20 flow, a token owner calls `approve` a single time on one of the 0x contracts and can then generate an unlimited amount of orders (intents to trade) off-chain with an associated signature. If we want to be able to post off-chain NFT orders in a similar way, we would require some way to approve all owned NFTs at once. Otherwise, an NFT owner will have to call `approve` every single time they want to trade a new NFT, removing most of the efficiency gains of having off-chain orders in the first place.



buhrmi commented a day ago • edited ▼

Hello guys, I wondered if there already is an effort to make a standard "shop" kind-of contract that works with the token-standard proposed here. I'm thinking of a contract or online shop that somehow can gain knowledge of the ERC-721 tokens and attach a price to it, making it purchasable, or run an auction on them.



dete commented 15 hours ago

Great to see all these new comments. Sadly, I've been a [touch busy](#) over the past couple of weeks... 😊

I've updated the first post as requested, and hope to digest the additional feedback and questions soon. I'm especially interested to get more info from **@eordano** and other implementers. For example, our implementation of `tokensOfOwner()` has been entirely non-functional on geth (and I assume other full nodes) ever since we crossed 100k NFTs or so. I'm curious at the gas cost implications of keeping track of a list of tokens for each owner inside `transfer` calls.



buhrmi commented 11 hours ago

Hmm, I think the `transferFrom` method can be removed since the `from` parameter depends on `tokenId` anyways. Instead, we can use `transfer` for approved `msg.sender` too.