Abstract

Contract metadata provides information that is not directly used for a contract's operation, whether about the contract's code (e.g. its interface, versioning) or the off-chain meaning of its contents (e.g. an artwork corresponding to an NFT). Currently, Tezos smart contracts lack a standard way to access such important data, fragmenting access to useful information that is needed for a scalabe integration experience by wallets, explorers, and applications.

To address this need and ease the integration, discoverability, and querying of Tezos smart contracts, we propose TZIP-016. TZIP-016 is a standard for encoding access to such smart contract metadata in JSON format either on-chain using tezos-storage or off-chain using IPFS or HTTP(S).

TZIP-016 defines:

- A basic structure to find *some* metadata in a contract's storage.
- An URI scheme to find data: on-chain (contract storage) or off-chain (web-services or IPFS).
- An extensible JSON format (JSON-Schema) to describe the metadata, it contains among other things:
    - provenance and authorship information,
    - references to other standards implemented,
    - off-chain “views” (Michelson functions to query the contract), and
    - custom extensions.
- optional entrypoints to validate metadata information.

The standard is meant to be extended/specialized by other TZIPs, for instance by adding fields to the JSON format of the metadata or imposing certain off-chain views. We intend to extend existing token APIs specified in TZIP-012 and TZIP-007 with such metadata and off-chain views using TZIP-016.

Table Of Contents

Introduction

This document defines a proposal for an interoperable method for encoding access to off-chain data from a Tezos contract (*KT1 account*). The method does not require a protocol change and seeks to minimize the use and impact of on-chain storage.

The goal is to allow smart contract authors, and wallet and indexers to agree on the location and the shape of a contract's metadata.

We define metadata as all information available on the contract that is not directly used for its operation, and *usually* only for off-chain consumption.

This includes authorship/copyright information, descriptions, and **off-chain-views**. Off-chain-views are queries (for now REST-API queries or Michelson functions) that anyone can run off-chain to obtain processed information on the current storage of a given contract.

Example Use-Case

For instance, an FA1.2 (i.e. ERC20-like) token contract may provide, among its metadata fields, a Michelson function "`get-balance`" which gives an account's current balance from the storage. The author of the contract can store this information on IPFS and add only an IPFS-URI to the contract's storage.

Conformity to the TZIP-016 (or a derivative) standard allows a wallet implementation to find the IPFS-URI and decode its contents to find the particular implementation of the `get-balance` off-chain-view in order to display a user's balance in its interface.

Definition of The Standard

In this section, we define the various components of the standard for human consumption (see annexes and reference implementations for more formal and machine-readable definitions).

Contract Storage

To provide a TZIP-016-compliant initial access-point to the metadata from a given on-chain contract (`KT1...` address) one must include a `%metadata` field in the contract-storage.

The field can be anywhere within the top-level tree of nested pairs forming the storage type (meaning that it cannot be inside an `or`, an `option`, a `map`, etc.) and must have the following type:

```
(big_map %metadata string bytes)
```

At least one value must be present:

- the one for the empty string key (`""`).
- the value must be a URI as specified in the following section which points to a JSON document as specified further below.

Unless otherwise-specified, the encoding of the values must be the direct stream of bytes of the data being stored. For instance, an URI starting with `http:` will start with the 5 bytes `0x687474703a` (`h` is `0x68`, `t` is `0x74`, etc.). There is no implicit conversion to Michelson's binary format (`PACK`) nor quoting mechanism.

Metadata URIs

URIs are used here first to locate metadata contents, but the format may be reused in other similar cases for instance in extensions to this specification.

See the specification of a URI's generic format: https://tools.ietf.org/html/rfc3986

In the context of this specification, valid schemes include:

- `http`/`https`: see RFC 7230.
- `ipfs`: see IPFS URIs specification, and documentation.
- `tezos-storage`: defined in the section below.
- `sha256`: defined in the section right after.

The `tezos-storage` URI Scheme

URIs that point at the storage of a contract, should provide the following information:

Host:

- Location of the contract pointed to, with the format `<address>.<network>`.
  - Valid addresses are base58check-encoded Tezos contract addresses: `KT1...`.
  - Valid networks are base58check-encoded Tezos chain identifiers `Net....` or known public network aliases: `mainnet`, `carthagenet`, `delphinet`, ...
- Example: `KT1QDFEu8JijYbsJqzoXq7mKvfaQQamHD1kX.mainnet` or `KT1QDFEu8JijYbsJqzoXq7mKvfaQQamHD1kX.NetXNfaaGTuJUGF`
- This is all optional, if contract address or network are not provided the defaults are "current" ones within a given context. If only one is present, it should be interpreted as a contract address within the current network.
- It is expected that given implementation of a URI resolver may not be able to handle every known network or even handle more than one; such cases should return/display a proper error message.

Path: a string used as key in the `%metadata` big-map of the contract. If the path starts with a `/` we remove it; only the first "slash" character is removed, if any (*rationale:* when URIs contain a host component, the path always starts with a `/`, cf. section 3.3 of RFC 3986). All other `/` characters are forbidden; if one needs to include such a character for the key in the big-map, it must be Percent-encoded (a.k.a. URL-encoded, cf. section 2.1), i.e. written as `%2F`.

Examples:

- `tezos-storage:hello`: in the current contract fetch the value at key `"hello"` from the `%metadata` big-map.
- `tezos-storage://KT1QDFEu8JijYbsJqzoXq7mKvfaQQamHD1kX/foo`: get the value at `foo` from the metadata big-map of the contract `KT1QDFEu8JijYbsJqzoXq7mKvfaQQamHD1kX` (on the current network).
- `tezos-storage://KT1QDFEu8JijYbsJqzoXq7mKvfaQQamHD1kX/%2Ffoo`: like the above, but the key is `/foo` (first `/` removed from the path, percent-encoded one is kept).
- `tezos-storage:hello/world` is invalid, while `tezos-storage:hello%2Fworld` is valid (the key is `hello/world`).

The `sha256` URI Scheme

This is a compound URI, the *host* must be understood as the SHA256 hash in hexadecimal format (preceded by `0x` as in Michelson) of the resource being pointed at by the path of the URI (which should be percent-

encoded to avoid ambiguity with `/` characters).

Example:

`sha256://0xeaa42ea06b95d7917d22135a630e65352cfd0a721ae88155a1512468a95cb750/https:%2F%2Ftezos.com`

Metadata JSON Format

We first define the format in a rather informal way; a JSON-Schema specification is provided as an annex to this document.

The metadata should be a valid JSON object (STD-90 / RFC-8259) with various top-level fields.

All top-level fields are optional, i.e. the empty object `{}` is valid metadata.

For compatibility, a compliant parser should ignore any extra fields it doesn't know about.

Reserved Fields

This standard defines a few top-level fields:

`"name"`:

- A single string, free format.
- It is recommended to have “name” strings help identifying the purpose of the contract.

`"description"`:

- A single string, free format.
- Preferably a set of proper *natural language* paragraphs.

`"version"`:

- A single string, free format.
- It is recommended to have version strings which attempt at uniquely identifying the exact Michelson contract, or at least its behavior, as a precision w.r.t the `name` field. Of course, none of that can be enforced.

`"license"`:

- An extensible object `{ "name": <string> , "details" : <string> }`, `"details"` being optional.
- It is recommended to use *de facto standard* short names when possible, see the Debian guidelines for instance.

`"authors"`:

- A list of strings.
- Each author should obey the `"Print Name <'contact'>"`, where the `'contact'` string is either an email address, or a web URI.

`"homepage"`:

- A single string, representing a “web” URL (e.g. HTTPS).
- The homepage is for human-consumption, it may be the location of the source of the contract, how to submit issue tickets, or just a more elaborate description.

`"source"`:

- An object `{ "tools": [<string>], "location": <string> }` describing the source code which was transformed or generated the Michelson code of the contract.
- `"tools"` is an informal list of compilers/code-generators/libraries/post-processors used to generate the originated code, if possible with version information.
- The goal is to attempt to provide enough information for interested parties to reproduce the Michelson from its source, or at least to inspect it.

`"interfaces"`:

- A list of strings.
- Each string should allow the consumer of the metadata to know which interfaces and behaviors the contract *claims* to obey (other than the obvious TZIP-016).
- In the case of standards defined as TZIPs in the present repository, the string should obey the pattern `"TZIP-<number><extras>"` where `<extras>` is additional information prefixed with a space character.
- Example: an FA2 contract would (at least) have an `"interfaces"` field containing `["TZIP-012"]` or `["TZIP-012 git 6544de32"]`.

`"errors"`:

- A list of “error translation” objects, which allow one to interpret error values output by the contract using the `FAILWITH` instruction. The interpretation is a larger data-structure, for instance, to provide to a wallet-user with a more understandable error message to act on (usually a `string` or `bytes` natural language text value). They are either:
  - static: `{ "error": <michelson>, "expansion": <michelson>, ... }`: where `<michelson>` is a Michelson expression in JSON (see also the following section on off-chain-views). The objects show the correspondences between `FAILWITH` values and *expanded* values.
  - dynamic: `{"view": <view-name>, ... }`: which means that one needs to call the off-chain-view `<view-name>` (a reference to a view in the `"views"` field below). The view's input type should be the one of the error value, and it should return the expanded structure.
  - both objects allow an extra field `"languages": [<lang1>, <lang2>, ...]`: a list of natural-language codes to filter-on if possible, the codes should be “IETF language tags” (cf. Wikipedia, and RFC-5646).
  - the objects can be redundant (incl. for various languages), implementations are expected to find the “best fit” according to their own priorities.

`"views"`:

- A list of off-chain-view objects, defined in the following section.

Example:

```
{
  "version": "foo.1.4.2",
```

```
  "license": { "name": "ISC" },
  "authors": [ "Seb Mondet <seb@mondet.org>" ],
  "source": {
     "tools": ["SmartPy dev-20201031", "Flextesa 20200921"],
     "location": "https://gitlab.com/smondet/fa2-
smartpy/-/blob/c05d8ff0/multi_asset.py"
  },
  "interfaces": [ "TZIP-012" ],
  "errors":[
    { "error": {"int": "42"},
      "expansion": { "string": "You did something wrong"},
      "languages": ["en"] },
    { "error": {"int": "42"},
      "expansion": { "bytes":
"0x7175656c7175652063686f7365206e276120706173206d61726368c3a9"},
      "languages": ["fr"] },
    { "view": "translateStringError" }
  ],
  "views": [
     // ... see below ...
  ]
  // ... potential extensions ...
}
```

Semantics of Off-chain Views

An off-chain view object has at least 2 fields:

- `"name"`; the canonical name of the query (as in function name, e.g. `"get-balance"`).
- `"description"`: a human readable description of the behavior of the view (optional field).
- `"implementations"`: a list of implementation objects: usable definitions of the views. Each implementation is a one-field object where the field name discriminates between various kinds of views. Below, this standard defines 2 of those kinds, `"michelsonStorageView"` and `"restApiQuery"`, further deriving standards may add new ones.
- `"pure"` (optional, default: `false`): a boolean "tag" advertising that the view should be considered *a (pure) function* of the storage of the contract and extra parameters passed to the view. I.e., that if no operation changes the storage of the contract, one can assume that the view returns always the same result for a given parameter. Examples of non-pure views are for instance queries which depend on the current time & date.

Example:

```
{
  "name": "get-allowance-for-user",
  "description": "Get the current allowance for a user of the contract.",
  "pure": "true",
  "implementations": [
     { "michelsonStorageView" : { /* ,,, see below ... */ } },
     { "restApiQuery" : { /* ,,, see below ... */ } },
     // ... potential extensions ...
```

```
    ]
}
```

Michelson Storage Views

The `"michelsonStorageView"` field is a JSON object describing a sequence of Michelson instructions to run on a pair formed by a given parameter and the storage of the contract being queried in order to leave the execution stack with the queried value. For this object we define 3 fields and a custom type "`michelson`" (see below) and an extra optional field:

- `"parameter"` (optional): an (annotated) Michelson type of the potential external parameters required by the view code; if the field is absent the view does not require any external input parameter.
- `"returnType"` (required): the type of the result of the view (i.e. for the value left on the stack); the type can also be annotated.
- `"code"` (required): the Michelson code expression implementing the view.
- `"annotations"`: a list of objects documenting the annotations used in the 3 above fields. These objects have two string fields `"name"`, the annotation string, and a human-readable blob of text `"description"`.
- `"version"` (optional): a string representing the version of Michelson that the view is meant to work with; Michelson versions *are* base58check-encoded protocol hashes.

The 3 "Michelson" fields have the same format, they are JSON values obeying the Michelson JSON format of the Tezos protocol (sometimes referred to as "Micheline" encoding). Only protocol-level primitives are allowed, the so-called "Michelson macros" are not).

Example:

```
{
  "parameter": {
    "prim": "pair", "args": [
      {"prim": "mutez", "annots": ["%amount"]},
      {"prim": "string", "annots": ["user"]}
    ]
  },
  "returnType": {"prim": "nat"},
  "code": [
    {"prim": "DUP"},
    {"prim": "DIP", "args": [
        [
          {"prim": "CDR"},
          {"prim": "PUSH", "args": [
    // ....
  ],
  "annotations": [
     { "name": "amount", "description": "The number of token in question." },
     { "name": "user", "description": "The token-user being referred to." },
     // ...
  ]
}
```

There are limitations on the contents of the `"code"` field:

- The following instructions must not be used: AMOUNT, CREATE_CONTRACT, SENDER, SET_DELEGATE, SOURCE, and TRANSFER_TOKENS
- The following instructions should be understood, as relative to the contract (and block) currently being queried: SELF, BALANCE, NOW, and CHAIN_ID.
- To simplify adoption by various implementations, in this first version of the specification, the instruction SELF should only be used before ADDRESS (i.e. only as `SELF; ADDRESS` should be used).
- All the above rules should apply to code contained in Michelson *lamdas* or serialized into `bytes` values (with `PACK`) but, as a recommendation, those techniques should be avoided in off-chain-views.

Rest API Views

The `"restApiQuery"` field is an object describing how to map the view to an Open API description of a REST-API.

- `"specificationUri"` (required): a string giving the location (URI) of the full Open API specification.
- `"baseUri"` (optional): The recommended `"server"` to use.
- `"path"` (required): The API path within the Open API specification that implements the view.
- `"method"` (optional, default: `"GET"`): The method used for the view.

Example:

```
{
  "specificationUri": "https://example.com/openapi/my-token",
  "baseUri": "https://example.com/my-token/v2",
  "path": "/allowances",
}
```

Optional `assertMetadata<hash>` Entrypoints

For the cases when a batched transaction requires assurances that a (portion of) the contract metadata has not changed at the time the batch-operation is included in a block, the contract implementation may provide one or more `assertMetadata<hash>` entrypoints where `<hash>` is any of the Michelson hash functions `SHA256`, `BLAKE2B`, or `SHA512`.

If included, the type of such an entrypoint must be:

```
(pair (string %key) (bytes %hash))
```

and behave as follows:

- If the corresponding hash of the value at key `%key` in the metadata `big_map` is equal to `%hash`, then do nothing and succeed.
- If the value is not present, call `FAILWITH` with the string `"NOT_FOUND"`,

- If the value is present but its hash is not equal to `%hash`, call `FAILWITH` with either `Unit` or with the correct hash if available.

Machine-Readable Specifications

In the present repository, `proposals/tzip-16/metadata-schema.json` is a JSON-Schema specification of the contents of the "Metadata JSON Format" described above. A few valid examples are available in the `proposals/tzip-16/examples/` directory.

How To "Derive" From TZIP-016

This proposal is meant to be extended and specialized by other standards. Here are some of the ways one can define TZIP-016 extensions:

- Defining new fields in the metadata-JSON.
- Making some of the metadata content mandatory, e.g. requiring the `"interfaces"` fields to be present, or requiring some off-chain-views to be implemented in Michelson with fixed parameter and return types.
- Using other *keys* of the `%metadata` big-map for storing particular data.
- Using the metadata-URI definition to locate other pieces of data.

Other aspects are better proposed as a backwards-compatible changes to TZIP-016. For instance, adding a new URI scheme to locate (meta)data can be better handled within this standard.

See also the future-work section for extensions that are already planned or in the works.

Known Implementations

The section will reference known implementations of (part of) TZIP-016:

- The work-in-progress merge-request `smondet/tezos!7` adds support in `tezos-client`: fetching and displaying metadata for any given contract as well as "calling" off-chain-views.
- The project `github.com/tqtezos/TZComet` (live at `tqtezos.github.io/TZComet`, previously known as *"Comevitz"*) provides validating metadata URI and JSON editors with various examples, a Michelson serialized data analyzer, and a (work-in-progress) metadata explorer which can fetch metadata URI and JSON contents.
- The Archetype language implementation has support in the compiler (cf. documentation).

Rationales / Design Choices

This section keeps track of some of the choices made in the development of the specification.

- *Use of the contract storage:* Another way of "storing" an URI on-chain could have been to use an "operation event" (a contract-call that is only used to record its payload on the blockchain without keeping anything in the contract's storage). This method is the cheapest in gas and storage but it requires an indexer for any implementation to be able to do anything. We consider this too much of a barrier for adoption. Moreover, we also want to leave the option of storing the metadata JSON *on-chain* (strongly requested by users).
- *The use of a big-map:* Once the contract-storage solution is assumed, we could have left the choice between `(bytes %metadata)` and `(big_map %metadata string bytes)`. The former's gain in type-

checking/deserialization gas seems negligible (we measured to around 250 units) compared to the extra complexity of the specification that it would incur.

- *The `string` in `(big_map string bytes)`:* The keys of the big-map are used for addressing values, including in the URI definition. The Michelson `string` type is a good practical choice for this; it has an obvious and readable concrete encoding: the string literal itself.
- *The `bytes` in `(big_map string bytes)`:* Michelson strings are limited in the characters one can encode (only ASCII printable and some whitespace), so to allow arbitrary values we need to use the `bytes` type. This includes the metadata JSON which, when stored on-chain, requires full UTF-8 capabilities (not supported by the `string` type).
- *The use of JSON-Schema instead of JSON_LD or other specifications:* Basic JSON(-Schema) was chosen because it is already pervasive in the Tezos ecosystem, including in the node RPCs defined in all the Mainnet protocols.

Future Work & Extensions

A few extensions and improvements of TZIP-016 are already planned or in progress:

- Augment/upgrade the TZIP-012 and TZIP-007 standards with metadata support.
- Specify *off-chain-events*: events are similar to off-chain-views, but they extract knowledge from (the sequence of) contract **calls**, like “balance updates”; cf. agora and this article.
- Specify *multi-contract off-chain views* which are off-chain-views defined over more than one contract-storages.

Copyright