

---

# Unique Network

*Release 0.0.1*

**unknown**

**Sep 06, 2021**



# CONTENTS

<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	Overview . . . . .	3
1.2	Getting Started . . . . .	3
1.3	JavaScript API . . . . .	7
1.4	Unity API . . . . .	31
1.5	.NET API . . . . .	32
1.6	Wallet Integration Guide . . . . .	32



Unique Network is the new home for NFT projects.



## CONTENTS

### 1.1 Overview

Unique Network blockchain in the Polkadot ecosystem can be seen as a foundation for standards and good practices serving for any software that uses or relates to NFT. The core components of Unique blockchain are:

- NFT Pallet
- Ink! Smart Contracts

Like ERC-721 Ethereum standard for smart contracts, NFT Pallet provides the basement for creating collections of NFTs, minting tokens, managing their ownership, and much more. The smart contracts module is included to handle any application logic that is unknown at the time of chain design.

The Unique Network aims to provide the feature rich and flexible configuration experience to its users. This includes multiple authorization levels, economic models that enable freemium application marketing, miscellaneous administration options, advanced spam protection. The goal is to cover the broad spectrum of NFT applications' development needs and provide maximum flexibility at low to affordable cost.

The Unique network is based on Polkadot Substrate, and works just like any blockchain based on substrate:

- It can be explored with Apps UI: <https://uniqueapps.usetech.com/>
- It can be integrated with any Polkadot API, such as JavaScript, C#, C++, or Python APIs, though this documentation mainly focuses on JavaScript API as the most up to date.

### 1.2 Getting Started

#### 1.2.1 Creating Accounts

Unique Network, like most blockchains, is based on accounts or addresses. An address can own NFTs or some Unique token. It can sign transactions to transfer these valuable assets to other addresses or to make some actions in Decentralized Apps (dApps). For example, an address can buy and sell NFTs on the NFT Market.


The typical Unique address looks like this:



```
5HEfXSCByZ9jgtrfSEQNnRSgRVf4wxiTyTzBME5xsjyNqak3
```

One way to create an address for yourself is to use [Polkadot{.js} browser extension for Chrome browser](#) or [Polkadot{.js} browser extension for Firefox](#).

Once it is installed, open the Polkadot{.js} extension and create the address. The extension will display the mnemonic seed. Make sure you save it securely because this is the only way to restore your address. This mnemonic seed can also


be used to sign transactions in JavaScript code. In the examples we use the mnemonic seed for Alice account (seed: “//Alice”), but you can replace it with your seed to work with TestNet or MainNet.


 Create an account 1/2 [Cancel](#)

 <unknown>  
5EPheBkzyPEZTRArnVg3iU5PFcugUvwzLbSCyYRAlk1BR8qL 


GENERATED 12-WORD MNEMONIC SEED:

tortoise display emerge strike inside sign cruel problem citizen raise  
bubble candy

 [Copy to clipboard](#)


 Please write down your wallet's mnemonic seed and keep it in a safe place. The mnemonic can be used to restore your wallet. Keep it carefully to not lose your assets.



I have saved my mnemonic seed safely.

Next step 

In the next step enter the name and password for the account. The password will be needed every time when you sign a transaction:



 Create an account 2/2 [Cancel](#)

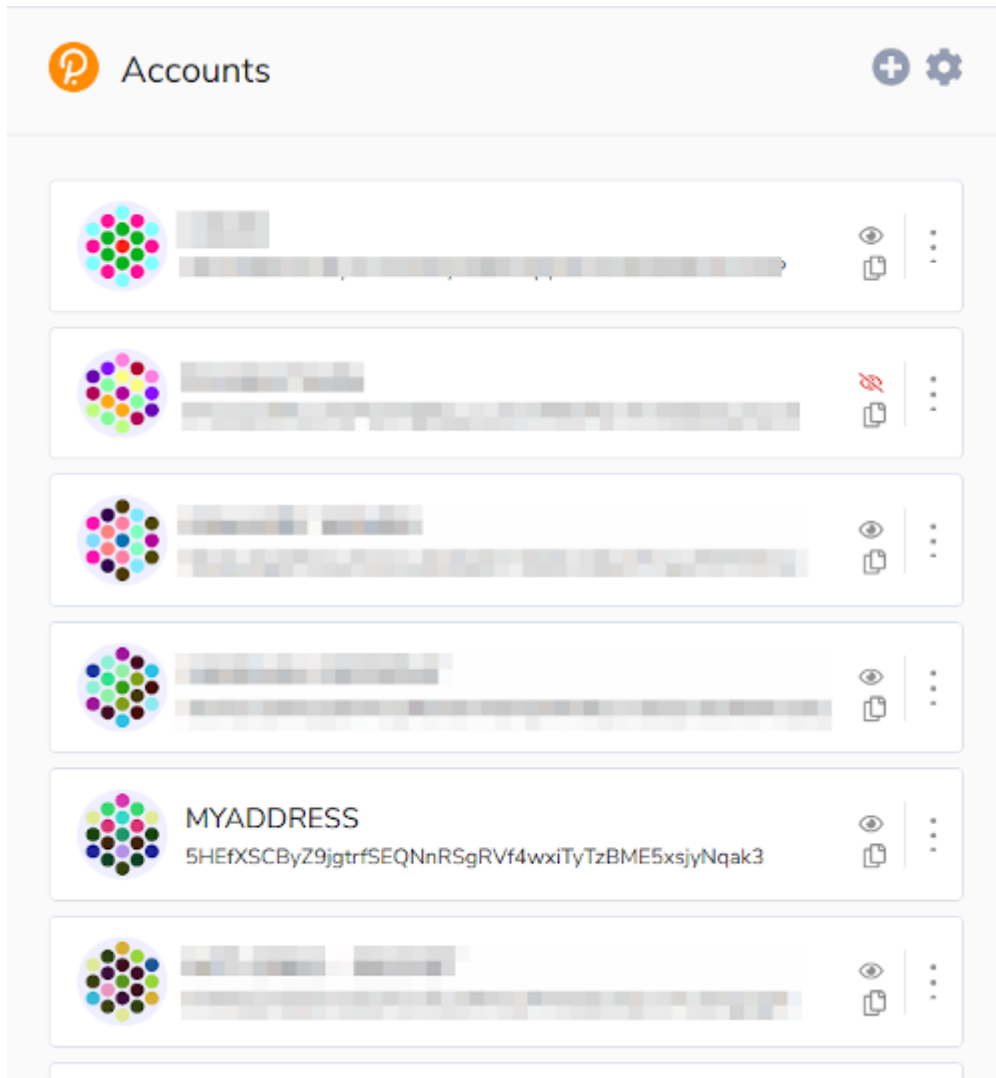
 <unknown>  
5Df2h9vAi3Hm6U8buWnV5DTnW7iJhxZWpvkWASNAZTcjKa5U 

A DESCRIPTIVE NAME FOR YOUR ACCOUNT

A NEW PASSWORD FOR THIS ACCOUNT

REPEAT PASSWORD FOR VERIFICATION

The new address will appear in the list:



Finally, you can open the [UniqueApps UI](#) to see your address.

## 1.2.2 Unique TestNet Faucet

In order to get transactions working on the TestNet, you will need some TestNet tokens.

You can get them from our Telegram bot: [@UniqueFaucetBot](#)

Once the transaction is processed, you may open the [UniqueApps UI](#) to see how your address' balance increased (make sure the UI is connected to the TestNet in settings page).

## 1.3 JavaScript API

### 1.3.1 Polkadot JS API

The [Polkadot JS API](#) is a constantly developed API for integration with Substrate based blockchains, which is maintained by Parity Inc.

This documentation does not focus on general features of this API, but mainly on using this API for integration with features of Unique Blockchain.

### 1.3.2 Installation

The Polkadot JS API is available as an npm package and can be included in *package.json* file as:

```
"@polkadot/api": "2.9.1",
```

### 1.3.3 Examples

The examples are provided for this documentation in the [examples folder](#). In order to execute them, install NodeJS 15, clone this repository and run an example (e.g. connect.js):

```
cd examples
npm install
node connect.js
```

### 1.3.4 Opening Connection

The Unique Network maintains public blockchain nodes to be used by clients for free. In order to connect to a client, you will need the public node URL and runtime types file that is located at [https://github.com/UniqueNetwork/nft\\_parachain/runtime\\_types.json](https://github.com/UniqueNetwork/nft_parachain/runtime_types.json).

The public node URL depends on the network that you would like to connect to:

Network	URL
TestNet 1.0	wss://unique.usetech.com
TestNet 2.0	wss://testnet2.uniquenetwork.io
MainNet	Coming soon...

Once you've got all parameters, connect to the node like this:

```
const { ApiPromise, WsProvider, Keyring } = require('@polkadot/api');
const rtt = require("./runtime_types.json");

const wsProvider = new WsProvider(public_node_url);

// Create the API and wait until ready
const api = await ApiPromise.create({
  provider: wsProvider,
  types: rtt
});
```

## 1.3.5 Collection Management

### Collection Properties

The following query can be used to get collection state:

```
await api.query.nft.collection(collectionId);
```

which returns an object like the following (for an NFT collection taken as example):

```
{
  Owner: 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY,
  Mode: {
    NFT: null
  },
  Access: Normal,
  DecimalPoints: 0,
  Name: [
    110,
    97,
    109,
    101,
    0
  ],
  Description: [
    100,
    101,
    115,
    99,
    114,
    105,
    112,
    116,
    105,
    111,
    110,
    0
  ],
  TokenPrefix: 0x70726566697800,
  MintMode: false,
  OffchainSchema: ,
  SchemaVersion: ImageURL,
  Sponsor: 5C4hrfjw9DjXZTzV3MwzrrAr9P1MJhSrvWGWqileSuyUpnhM,
  SponsorConfirmed: false,
  Limits: {
    AccountTokenOwnershipLimit: 10,000,000,
    SponsoredMintSize: 4,294,967,295,
    TokenLimit: 4,294,967,295,
    SponsorTimeout: 14,400
  },
  VariableOnChainSchema: ,
  ConstOnChainSchema:
}
```

**Fields**

- Owner - Collection owner
- Mode - type of collection (NFT, Fungible (ERC-20), or ReFungible)
- Access - Normal (for public access) or WhiteList (for restricted access)
- DecimalPoints - Number of decimal digits for value (only for Fungible collections)
- Name - Collection name (up to 64 UTF-16 characters)
- Description - Collection description (up to 256 UTF-16 characters)
- TokenPrefix - Token name as displayed in wallets (up to 16 UTF-8 characters)
- MintMode - True, if anyone is allowed to mint. False otherwise. See *setMintPermission*
- SchemaVersion - see *Data Schema*
- OffchainSchema - see *Data Schema*
- VariableOnChainSchema - see *Data Schema*
- ConstOnChainSchema - see *Data Schema*
- Sponsor - see *Economic Models*
- SponsorConfirmed - see *Economic Models*
- Limits - see *setCollectionLimits*

**createCollection****Description**

This method creates a Collection of NFTs. Each Token may have multiple properties encoded as an array of bytes of certain length. The initial owner and admin of the collection are set to the address that signed the transaction. Both addresses can be changed later.

**Permissions**

- Anyone

**Parameters**

- collectionName: UTF-16 string with collection name (limit 64 characters)
- collectionDescription: UTF-16 string with collection description (limit 256 characters)
- tokenPrefix: UTF-8 string with token prefix, limit 16 characters
- collectionType:
  - 0 - Invalid (collection does not exist, if type is 0)
  - 1 - NFT. All items in ItemList are unique and indivisible (decimalPoints parameter must be 0). Item IDs are unique, and one item may only be owned by one address.
  - 2 - Fungible. Collection does not have custom data associated with token (custom data size parameter must be 0). All Item IDs are the same and all that is recorded in ItemList in value field is the owner address and owned amount. The value is fixed point decimal with decimalPoints set as in the parameter to this method.
  - 3 - Re-Fungible. Custom data is allowed, but Items IDs are not unique. One item may be owned by more than one address. Value in ItemList entry corresponds to the owned portion of token. Value is an integer number and corresponds to the number of owned pieces.

- decimalPoints: Decimal points to be used in token amounts. If set to 0, tokens are indivisible.

### Events

- CollectionCreated
  - CollectionID: Globally unique identifier of newly created collection.
  - Owner: Collection owner

### Code example:

```
await api.tx.nft.createCollection();
```

More complete examples can be found here: [https://github.com/UniqueNetwork/unique-docs/blob/master/examples/token\\_management.js](https://github.com/UniqueNetwork/unique-docs/blob/master/examples/token_management.js)

## changeCollectionOwner

### Description

Change the owner of the collection

### Permissions

- Collection Owner

### Parameters

- CollectionId - ID of the collection to change owner for
- NewOwner - new collection owner

## destroyCollection

### Description

DANGEROUS: Destroys collection and all NFTs within this collection. Users irrecoverably lose their assets and may lose real money.

### Permissions

- Collection Owner

### Parameters

- CollectionId - ID of the collection to destroy

## setVariableMetaData

### Description

Update token custom data (the changeable part).

### Permissions

Permissions (whether a user can change this metadata) are set by *setmetadataupdatepermissionflag* method. The default is:

- Collection Owner
- Collection Admin

- Current NFT Owner

**Parameters**

- CollectionID: ID of the collection
- ItemID: ID of NFT to set metadata for

**addCollectionAdmin****Description**

NFT Collection can be controlled by multiple admin addresses (some which can also be servers, for example). Admins can issue and burn NFTs, as well as add and remove other admins, but cannot change NFT or Collection ownership.

This method adds an admin of the Collection.

**Permissions**

- Collection Owner
- Collection Admin

**Parameters**

- CollectionID: ID of the Collection to add admin for
- Admin: Address of new admin to add

**removeCollectionAdmin****Description**

Remove admin address of the Collection. An admin address can remove itself. List of admins may become empty, in which case only Collection Owner will be able to add an Admin.

**Permissions**

- Collection Owner
- Collection Admin

**Parameters**

- CollectionID: ID of the Collection to remove admin for
- Admin: Address of admin to remove

**setPublicAccessMode****Description**

Toggle between normal and white list access for the methods with access for “Anyone”.

**Permissions**

Collection Owner

**Parameters**

- CollectionID: ID of the Collection to remove admin for
- Mode

- 0 = Normal
- 1 = White list

### **addToWhiteList**

#### **Description**

Add an address to white list.

#### **Permissions**

- Collection Owner
- Collection Admin

**Parameters** \* CollectionID: ID of the Collection \* Address

### **removeFromWhiteList**

#### **Description**

Remove an address from white list.

#### **Permissions**

- Collection Owner
- Collection Admin

#### **Parameters**

- CollectionID: ID of the Collection
- Address

### **setMintPermission**

#### **Description**

Allows Anyone to create tokens if:

- White List is enabled, and
- Address is added to white list, and
- This method was called with True parameter

#### **Permissions**

- Collection Owner

#### **Parameters**

- CollectionID: ID of the Collection to add admin for
- MintPermission: Boolean parameter. If True, allows minting to Anyone with conditions above.



## setCollectionLimits

### Description

Sets some collection limits and starts enforcing them immediately (with no exception for collection owner or admins). By default the collection limits are not set, so for example, the number of items that an address can own is not limited. When the limits are set, the current number of owned items will be checked, and if it already exceeds the limit, the transaction will fail. After the limits are set, they start being enforced.

Note that some bounds are also set by the global chain limits (see *setChainLimits*). The more restrictive limits will always apply.

- *AccountTokenOwnershipLimit* - Maximum number of tokens that one address can own. Default value is the maximum value of 10,000,000,000,000. When the number of tokens owned by a single address reaches this number, no more tokens can be transferred or minted to this address.
- *SponsoredMintSize* - maximum byte size of custom NFT data that can be sponsored when tokens are minted in sponsored mode. If the amount of custom data is greater than this parameter when tokens are minted, then the transaction sender will pay transaction fees when minting tokens.
- *TokenLimit* - total amount of tokens that can be minted in this collection. Default value is the maximum value of 10,000,000,000,000. When the limit is set, the NFT pallet will check if the number of minted tokens is less or equal than the parameter value. If the number of minted tokens is greater than this number, the transaction will fail. This limit is designed to facilitate token scarcity. So, it can only be set to a lower value than previous (or if previous value is default).
- *SponsorTimeout* - Time interval in blocks that defines once per how long a non-privileged user transfer or mint transaction can be sponsored. Default value is 14400 (24 hrs), allowed values are from 0 (not limited) to 10,368,000 (1 month).
- *OwnerCanTransfer* - Boolean value that tells if collection owner or admins can transfer or burn tokens owned by other non-privileged users. This is a one-way switch: If it is ever disabled (set to *false*), it cannot be re-enabled (set back to *true*).
- *OwnerCanDestroy* - Boolean value that tells if collection owner can destroy it. This is a one-way switch: If it is ever disabled (set to *false*), it cannot be re-enabled (set back to *true*).
- *VariableMetadataSponsoringRateLimit* - Time interval in blocks that defines once per how long a non-privileged user transaction to update variable metadata can be sponsored. Default value is 0 (never sponsored), allowed values are from 0 (never sponsored) to 10,368,000 (1 month).

### Permissions

- Collection Owner

### Parameters

- *collectionId*: ID of the collection to set limits for
- *CollectionLimits* structure (see the description of fields above)

### setTransferEnabledFlag

#### Description

Enable or disable transfers in a collection.

#### Permissions

- Collection Owner

#### Parameters

- CollectionID: ID of the Collection to add admin for
- TransferFlag: Boolean parameter. If True, allows transfers, otherwise token transfers are frozen

### setMetadataUpdatePermissionFlag

#### Description

Set the permissions for token metadata updates. By default, the variable NFT metadata can be updated by a user who owns the token, but this behavior can be changed and set to one of the following:

- Item\_owner: Default, user who owns the token.
- Admin: Only collection owner and admins can change variable metadata. A smart contract may also be made an admin in order to change token properties trustlessly.
- None: Nobody can update variable metadata, including the token and collection owner. This option is irreversible. Once it is set, the variable token metadata becomes permanent in this collection.

#### Permissions

- Collection Owner

#### Parameters

- CollectionID: ID of the Collection to add admin for
- PermissionFlag: Permission flag, see description above

## 1.3.6 Token Management

### createItem (Mint)

#### Description

This method creates a concrete instance of NFT, Fungible, or ReFungible Collection created with *createCollection* method.

#### Permissions

- Collection Owner
- Collection Admin
- Anyone, if
  - White List is enabled, and
  - Address is added to white list, and
  - MintPermission is enabled (see setMintPermission method)

## Parameters

- CollectionID: ID of the collection
- Owner: Address, initial owner of the token
- Properties: Depends on collection type
  - NFT: Arrays of bytes that contain NFT properties. Since NFT Module is agnostic of properties' meaning, it is treated purely as an array of bytes.
    - \* const\_data: Immutable properties
    - \* variable\_data: Mutable properties
  - Fungible: Amount to create (multiplied by 10 to the decimalPoints power. E.g. if decimalPoints equals 2, number 301 creates 3.01 tokens)
  - ReFungible:
    - \* const\_data: Immutable properties
    - \* variable\_data: Mutable properties
    - \* pieces: Number of pieces this token is divided into

## Events

- **ItemCreated**
  - CollectionID: ID of collection
  - ItemId: Depends on the collection type:
    - \* NFT: Identifier of newly created NFT. which is unique within the Collection, so the NFT is uniquely identified with a pair of values: CollectionId and ItemId.
    - \* Fungible: Item IDs are not used, so the value is just 0
    - \* ReFungible: Same as NFT
  - Recipient: Address that receives token

## Code example:

```
const nftItemId = await createItem(
  api,
  alice,
  nftCollectionId,
  // Token receiver
  alice.address,
  {
    nft: {
      // Arbitrary data assigned to token
      const_data: [1, 2, 3, 4],
      // Variable data can be set later with setVariableMetadata
      variable_data: [1, 2, 3, 4],
    },
  },
);
```

More complete examples can be found here: [https://github.com/UniqueNetwork/unique-docs/blob/master/examples/token\\_management.js](https://github.com/UniqueNetwork/unique-docs/blob/master/examples/token_management.js)

### createMultipleItems

#### Description

This method creates multiple instances of NFT, Fungible, or ReFungible Collection created with *createCollection* method.

#### Permissions

- Collection Owner
- Collection Admin
- Anyone, if
  - White List is enabled, and
  - Address is added to white list, and
  - MintPermission is enabled (see setMintPermission method)

#### Parameters

- CollectionID: ID of the collection
- Owner: Address, initial owner of all tokens created in this transaction
- Items: Array of items to create. Each single item is described by properties as in `createItem`_`` method

#### Events

One *ItemCreated* event is emitted for each created token

- **ItemCreated**
  - CollectionID: ID of collection
  - ItemId: Depends on the collection type:
    - \* NFT: Identifier of newly created NFT. which is unique within the Collection, so the NFT is uniquely identified with a pair of values: CollectionId and ItemId.
    - \* Fungible: Item IDs are not used, so the value is just 0
    - \* ReFungible: Same as NFT

### burnItem

#### Description

This method destroys a concrete instance of NFT.

#### Permissions

- Collection Owner
- Collection Admin
- Current NFT Owner

#### Parameters

- CollectionID: ID of the collection
- ItemID: ID of NFT to burn
  - Non-Fungible Mode: Required

- Fungible Mode: Ignored
- Re-Fungible Mode: Required
- Value: Amount to burn
  - Non-Fungible Mode: Ignored (only the whole token can be burned)
  - Fungible Mode: Must specify transferred amount
  - Re-Fungible Mode: Ignored (the owned portion is burned completely)

#### Events

- **ItemDestroyed**
  - CollectionID: ID of collection
  - ItemId: Identifier of burned NFT

#### Code example:

```
await burnItem(api, alice, nftCollectionId, nftItemId, 1);
```

More complete examples can be found here: [https://github.com/UniqueNetwork/unique-docs/blob/master/examples/token\\_management.js](https://github.com/UniqueNetwork/unique-docs/blob/master/examples/token_management.js)

### Getting Token Information

In order to get the NFT or Re-fungible token information, one should use

- *api.query.nft.nftItemList* query for Non-Fungible items
- *api.query.nft.reFungibleItemList* query for Re-Fungible items

#### Parameters

- CollectionID: Id of collection
- ItemID: token Id

The API will return the JSON structure in the following format that contains

```
{
  Collection: 4,
  Owner: 5FZeTmbZQZsJcyEevjGVK1HHkcKfWBYxWpbgEfffQ2M1SqAnP,
  Data: 0x0001000311ffffffffffffffffffffffffffffffffffff
}
```

### 1.3.7 Token Ownership and Transfers

This group of methods allows managing NFT ownership.

### Getting BalanceOf

In order to get the NFT or Re-fungible balance for an address, one should use *api.query.nft.balance*

#### Parameters

- CollectionID: Id of collection
- AccountId: user address

### Getting Address Tokens

In order to get the list of NFT or Re-fungible tokens that are owned by a single address, one should use *api.query.nft.addressTokens*

#### Parameters

- CollectionID: Id of collection
- AccountId: user address

### Transfer Checks

This algorithm is used to check if the address can transfer, approve, transferFrom, and burn a token:

1. Check ownership and/or approvals (If not -> Error. If yes -> go next.)
  1. Transfer, Approve, and Burn: Check if the sender owns the token, or
  2. TransferFrom: Check if the sender is approved to transfer this token. Collection Owner, Admins, and this token owner are always approved.
2. Check if the sender is the collection owner or an admin. If yes -> Allow transaction, no extra checks needed. If no -> go next.
3. Check if White List mode is enabled. If no -> Allow transaction, no extra checks needed. If yes -> go next.
4. Check if the sender is in the white list. If yes -> Allow transaction, no extra checks needed. If no -> Error.

### transfer

#### Description

Change ownership of the token.

#### Permissions

- Collection Owner
- Collection Admin
- Current NFT owner

#### Parameters

- Recipient: Address of token recipient
- CollectionId: ID of collection
- ItemId: ID of the item
  - Non-Fungible Mode: Required

- Fungible Mode: Ignored
- Re-Fungible Mode: Required
- Value (Optional): Amount to transfer
  - Non-Fungible Mode: Ignored
  - Fungible Mode: Must specify transferred amount
  - Re-Fungible Mode: Must specify transferred portion (between 0 and 1)

#### Events

- Transfer
  - Collection ID
  - Token ID
  - Sender address
  - Recipient address
  - Amount (always 1 for NFT)

#### **transferWithData (not yet available)**

##### **Description**

This ERC-721 compatibility method is not yet implemented.

Same as Transfer with extra parameter: Data, an array of bytes. Data will be emitted in an event.

##### **Permissions**

Same as transfer

##### **Parameters**

- Recipient: Address of token recipient
- CollectionId: ID of collection
- ItemId: ID of the item
- Data: Data to be included in the transaction

#### **transferFrom**

##### **Description**

Change ownership of a NFT on behalf of the owner. See Approve method for additional information. After this method executes, the approval is removed so that the approved address will not be able to transfer this NFT again from this owner.

##### **Permissions**

- Collection Owner
- Collection Admin
- Current NFT owner
- Address approved by current NFT owner

### Parameters

- Sender: Address that owns token
- Recipient: Address of token recipient
- CollectionId: ID of collection
- ItemId: ID of the item

### Events

- Transfer
  - Collection ID
  - Token ID
  - Sender address
  - Recipient address
  - Amount (always 1 for NFT)

### **transferFromWithData (not yet available)**

#### Description

This ERC-721 compatibility method is not yet implemented.

Same as TransferFrom with extra parameter: Data, an array of bytes. Data will be emitted in an event.

#### Permissions

Same as TransferFrom

#### Parameters

- Sender: Address that owns token
- Recipient: Address of token recipient
- CollectionId: ID of collection
- ItemId: ID of the item
- Data: Data to be included in the transaction

### **approve**

#### Description

Set, change, or remove approved address to transfer the ownership of the token. The Amount value must be between 0 and owned amount or 1 for NFT.

#### Permissions

- Collection Owner
- Collection Admin
- Current NFT owner

#### Parameters

- Spender: Address that is approved to transfer this token



- CollectionId: ID of collection
- ItemId: ID of the item
- Amount:
  - Non-Fungible Mode: Required, must be 1 (for approval) or 0 (for disapproval).
  - Fungible Mode: Required, amount to add to approved amounts for the Spender or 0 (to remove approvals)
  - Re-Fungible Mode: Required, amount to add to approved amounts for the Spender or 0 (to remove approvals)

### Events

- Approved
  - Collection ID
  - Token ID
  - Sender address
  - Spender address
  - Amount (always 1 for NFT)

### setApprovalForAll (not yet available)

#### Description

This ERC-721 compatibility method is not yet implemented.

Sets or unsets the approval of a given address (operator). An operator is allowed to transfer all tokens of the sender on their behalf. Unlike single approvals, approvals granted using this method don't reset after transfers.

#### Permissions

- Collection Owner
- Collection Admin
- Current NFT owner

#### Parameters

- CollectionId: ID of the collection
- Approved: True or False

### Getting Approvals

The current approvals may be read with *api.query.nft.approvedList*. It returns the list of addresses, approved for the given token.

#### Parameters

- CollectionId: ID of collection
- ItemId: ID of the item

### **batchTransfer**

This is an ERC-1155 compatibility method. Not implemented yet

### **batchApproval**

This is an ERC-1155 compatibility method. Not implemented yet

### **batchTransferFrom**

This is an ERC-1155 compatibility method. Not implemented yet

### **safeBatchTransfer**

This is an ERC-1155 compatibility method. Not implemented yet

### **safeBatchTransferFrom**

This is an ERC-1155 compatibility method. Not implemented yet

## **1.3.8 Data Schema**

### **setSchemaVersion**

#### **Description**

Set schema standard to one of:

- ImageURL (Image URL only, just like in TestNet 1.0)
- Unique
- OpenSea
- Tezos TZIP-16 (<https://gitlab.com/tzip/tzip/-/blob/master/proposals/tzip-16/tzip-16.md>)

The data schema is used by NFT wallets in order to display the token metadata, as well as offchain token data (such as images, etc.) correctly in the wallet. *Unique Wallet* currently supports *ImageURL* and *Unique* formats.

#### **Image URL**

This schema format assumes saving the image URL template in *constOnChainSchema*. The image template allows NFT wallets to reconstruct the full image URL for each token using its ID. The URL template can contain {id} placeholder that will be replaced with the actual token ID when the image URL is reconstructed.

Example:

```
https://ipfs-gateway.usetech.com/ipns/QmaMtDqE9nhMX9RQLTpaCboqg7bqkb6Gi67iCKMe8NDpCE/  
↪ images/punks/image{id}.png
```

#### **Unique**

The *Unique* format allows NFT wallets to decode on-chain token metadata and access off-chain data. This format is currently evolving and may update in the future. It supports three schemas: constant on-chain, variable on-chain, and off-chain. The schema is the JSON string that contains information about how to access and decode token metadata.

In case of off-chain metadata, the data is accessed at a 3rd party or an IPFS URL. URLs may contain the {id} placeholder that will be replaced by the wallet in order to reconstruct the URL for that resource. Currently the Unique Wallet only supports “metadata” entry (just like in the example below). The JSON object returned by the metadata endpoint must contain “image” key with image URL value.

In case of on-chain metadata, the data is binary (i.e. an array of bytes), and it is encoded with protobuf codec, so the schema shows how to deserialize that binary on-chain data into human readable entries. The off-chain schema has the same format as .proto files in protobuf serializer (see example below). The package name should always be equal to *onchainmetadata*, and the root object should always be named *NFTMeta*. In order to encode large strings for converting enum values in multiple languages, one can use JSON transaction object in the single line comments before the enum value in the enum definition (see the example).

Example for const or variable on-chain that is used by SubstraPunks (shortened version):

```
package onchainmetadata;
syntax = "proto3";

enum Gender {
  /// {"cn": "", "en": "Male", "ru": ""}
  Male = 0;
  /// {"cn": "", "en": "Female", "ru": ""}
  Female = 1;
};

enum PunkTrait {
  /// {"cn": "", "en": "Black Lipstick", "ru": " "}
  BLACK_LIPSTICK = 0;
  /// {"cn": "", "en": "Red Lipstick", "ru": " "}
  RED_LIPSTICK = 1;
  /// {"cn": "", "en": "Smile", "ru": ""}
  SMILE = 2;
  /// {"cn": "", "en": "Teeth Smile", "ru": " "}
  TEETH_SMILE = 3;
  /// {"cn": "", "en": "Purple Lipstick", "ru": " "}
  PURPLE_LIPSTICK = 4;
  /// {"cn": "", "en": "Nose Ring", "ru": " "}
  NOSE_RING = 5;
  /// {"cn": "", "en": "Asian Eyes", "ru": " "}
  ASIAN_EYES = 6;
  /// {"cn": "", "en": "Sunglasses", "ru": " "}
  SUNGLASSES = 7;
};

/// This is the root object of the schema, it will always be called "NFTMeta"
message NFTMeta {
  required Gender gender = 1;
  repeated PunkTrait traits = 2;
}
```

Example for off-chain schema:

```
{
  "metadata": "https://ipfs-gateway.usetech.com/ipns/
  ↪QmaMtDqE9nhMX9RQLTpaCboqg7bqkb6Gi67iCKMe8NDpCE/metadata/token{id}"
}
```

Example of data returned from metadata endpoint for token ID 1:

```
{
  "image" : "https://ipfs-gateway.usetech.com/ipns/
  →QmaMtDqE9nhMX9RQLTpaCboqg7bqkb6Gi67iCKMe8NDpCE/images/punks/image1.png"
}
```

This [protobuf example](#) shows how to decode the substrapunk schema using JavaScript.

### Permissions

- Collection Owner
- Collection Admin

### Parameters

- CollectionID: ID of collection
- SchemaVersion: enum

## setOffchainSchema

### Description

Set off-chain data schema. In the initial version of NFT parachain the schema will only reflect image URL. The `{id}` substring will be parsed to reflect the NFT id.

For example, the schema string for CryptoKitties will look like this:

```
https://img.cryptokitties.co/0x06012c8cf97bead5deae237070f9587f8e7a266d/{id}.png
```

Next version of the token data schema is split into three methods: `SetOffchainSchema`, `SetConstOnChainSchema`, and `SetVariableOnChainSchema`, as well as a chain variable: `SchemaVersion`, which will return the value corresponding to the metadata standard being used. If `SchemaVersion` is not present in the chain, it means this is still the TestNet 1.0 and there is no on-chain schema yet implemented in it.

The schema must contain the image and page fields, which should use `{id}` placeholder that will be replaced by wallets with the actual token ID in order to get the token page and image URLs. Also, there is an optional “audio” field that contains audio file URL associated with the tokens. The schema will be parsed by 3rd party wallets, but not at the moment of setting the schema.

Example:

```
{
  "image": "https://example.com/images/{id}",
  "page": "https://example.com/nft/{id}",
  "audio": "https://example.com/audio/{id}"
}
```

### Permissions

- Collection Owner
- Collection Admin

### Parameters

- CollectionID: ID of collection
- Schema: String representing the offchain data schema

## setConstOnChainSchema

### Description

Set the on-chain schema (string in JSON-schema format) that describes permanent token fields.

This schema describes the serialization of non-changeable token fields. Serialization algorithm depends on the version of schema selected in *setSchemaVersion* . Unique schema uses Google protobuf for serialization, which is described in *setSchemaVersion* .

The schema will be parsed by 3rd party wallets, but it is not validated at the moment when it is set.

Example: see example in *setSchemaVersion*

### Permissions

- Collection Owner
- Collection Admin

### Parameters

- CollectionID: ID of collection
- Schema: String representing the offchain data schema

## setVariableOnChainSchema

### Description

Same as Const on-chain schema, except sets the variable schema. Also, requires name and size of each field and is required to match the total variable data size.

### Permissions

- Collection Owner
- Collection Admin

### Parameters

- CollectionID: ID of collection
- Schema: String representing the offchain data schema

## Getting Data Schemas

In order to get a data schema for the collection, one should use following query: *api.query.nft.collection*. The response to the query is the JSON object that contains schemas information in fields *OffchainSchema*, *VariableOnChainSchema*, and *ConstOnChainSchema*:

```
{ Owner: 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY, Mode: {
  NFT: null
}, Access: Normal, DecimalPoints: 0, Name: [
  0
], Description: [
  0
]
```

```
], TokenPrefix: 0x3000, MintMode: false, OffchainSchema: "", Sponsor:  
5C4hrfjw9DjXZTzV3MwzrrAr9P1MJhSrvWGWqi1eSuyUpnhM, SponsorConfirmed: false,  
VariableOnChainSchema: "", ConstOnChainSchema: ""  
}
```

### Parameters

- CollectionID: Id of collection

### Code Example

```
await api.query.nft.collection(collectionId);
```

## 1.3.9 Economic Models

The Unique Network allows sponsoring user transactions for NFT collections and smart contracts. When collection (or smart contract) is sponsored, all their users need is to have the Unique wallet and address, but they don't need to have any Unique balance on the wallet. This feature removes the extra friction for the end user and creates nice flawless user experience.

### setCollectionSponsor

#### Description

Setting collection sponsor is the 2-step process. This method is the step 1: Set the sponsor address. The sponsor will need to confirm the sponsorship using *confirmSponsorship* method before the sponsoring begins.

#### Permissions

- Collection Owner

#### Parameters

- CollectionID: ID of collection
- Sponsor: Sponsor address

### confirmSponsorship

#### Description

Setting collection sponsor is the 2-step process. This method is the step 2: Confirm sponsorship. The sponsor needs to confirm the sponsorship so that the collection owners cannot attack the addresses they are not related with.

#### Permissions

- Collection Sponsor

#### Parameters

- CollectionID: ID of collection

## removeCollectionSponsor

### Description

Disable sponsoring and switch back to pay-per-own-transaction model.

### Permissions

- Collection owner

### Parameters

- CollectionID: ID of collection

## Enabling Contract Sponsoring (EVM)

### Description

In order to enable contract sponsoring on EVM (Ethereum) contract, web3 library needs to be used because EVM contracts are deployed using ETH RPC interface, so the owner of the EVM contract is an Ethereum address. This short example demonstrates how to enable sponsoring for a contract with address stored in *myContractAddress* variable:

```
import Web3 from 'web3';
...
const helpers = new web3.eth.Contract(contractHelpersAbi as any,
  ↪ '0x842899ECF380553E8a4de75bF534cdf6fBF64049', {from: caller, ...GAS_ARGS});
await helpers.methods.toggleSponsoring(myContractAddress, true).send({from: owner});
await helpers.methods.toggleAllowlist(myContractAddress, true).send({ from: owner });
```

Note that *helpers.methods.toggleAllowlist* call is also included in this example because enabling allow list is required in order for sponsoring to work (as a security measure). Read more about this below.

### Permissions

- Address that deployed smart contract

### Parameters

- contractAddress: Address of the contract to sponsor
- enable: Boolean flag to enable or disable smart contact self-sponsoring

## enableContractSponsoring (Ink!)

### Description

Note: The Ink! smart contracts are currently disabled.

Enable the Ink! smart contract to pay for its own transaction using its endowment. Can only be called by the contract owner, i.e. address that deployed this smart contract. The sponsoring will only start working after the rate limit is set with `setContractSponsoringRateLimit-ink`_``.

### Permissions

- Address that deployed smart contract

### Parameters

- contractAddress: Address of the contract to sponsor
- enable: Boolean flag to enable or disable smart contact self-sponsoring

### Settings Contract Sponsoring Rate Limit (EVM)

#### Description

Set the rate limit for contract sponsoring. The default value for the rate limit is 7200 blocks, i.e. one day. If set to the number B (for blocks), the transactions will be sponsored with a rate limit of B, i.e. fees for every transaction sent to this smart contract will be paid from contract balance if there are at least B blocks between such transactions. Nonetheless, if transactions are sent more frequently, the fees are paid by the sender.

This short example demonstrates how to set sponsoring rate limit of one transaction per 1234 blocks for a contract with address stored in *myContractAddress* variable:

```
import Web3 from 'web3';
...
const helpers = new web3.eth.Contract(contractHelpersAbi as any,
  ↪ '0x842899ECF380553E8a4de75bF534cdf6BF64049', {from: caller, ...GAS_ARGS});
await helpers.methods.setSponsoringRateLimit(myContractAddress, 1234).send({from: owner}
  ↪);
```

#### Permissions

- Address that deployed smart contract

#### Parameters

- contractAddress: Address of the contract to sponsor
- rate\_limit: Number of blocks to wait until the next sponsored transaction is allowed

### setContractSponsoringRateLimit (Ink!)

Note: The Ink! smart contracts are currently disabled.

#### Description

Set the rate limit for contract sponsoring. If not set (has the default value of 0 blocks), the sponsoring will be disabled. If set to the number B (for blocks), the transactions will be sponsored with a rate limit of B, i.e. fees for every transaction sent to this smart contract will be paid from contract endowment if there are at least B blocks between such transactions. Nonetheless, if transactions are sent more frequently, the fees are paid by the sender.

#### Permissions

- Address that deployed smart contract

#### Parameters

- contractAddress: Address of the contract to sponsor
- rate\_limit: Number of blocks to wait until the next sponsored transaction is allowed



## Sponsor Security

Sponsoring smart contracts is tricky. Users can generate addresses very quickly because creating an address is as simple as generating a random 64-byte sequence. So, it is really hard to prevent someone from making very many smart contract calls if they are sponsored. But the sponsor funds need to be protected.

One way to protect funds is to introduce severe rate limits globally, i.e. for all users of the smart contract, but it also degrades the user experience, especially if there are malicious players who race for free contract calls.

The ``setContractSponsoringRateLimit-ink`_` only limits the call rate for each address, so it is designed to be used with White Lists, enabled by ``toggleContractWhiteList`_`, when the number of addresses is limited.

So the quick recipe for secure smart contract sponsoring is:

RATE LIMIT + WHITE LIST

The contract owner (address that deployed it) can add user addresses to the white lists using ``addToContractWhiteList-ink`_` method. For a dApp this can be combined with user registration, when the account is confirmed (or captcha or KYC is passed, for example).

## Toggle Contract Allow List (EVM)

### Description

In order to enable allow list on an EVM (Ethereum) contract, web3 library needs to be used because EVM contracts are deployed using ETH RPC interface, so the owner of the EVM contract is an Ethereum address. This short example demonstrates how to enable allow lists for a contract with address stored in `myContractAddress` variable:

```
import Web3 from 'web3';
...
const helpers = new web3.eth.Contract(contractHelpersAbi as any,
  ↪ '0x842899ECF380553E8a4de75bF534cdf6fBF64049', {from: caller, ...GAS_ARGS});
await helpers.methods.toggleAllowlist(myContractAddress, true).send({ from: owner });
```

### Permissions

- Address that deployed smart contract

### Parameters

- `contractAddress`: Address of the EVM contract
- `enable`: Boolean that tells to either enable (if true) or disable (if false) the allow list for that EVM smart contract

## toggleContractWhiteList (Ink!)

### Description

Enable the white list for a contract. If enabled, only addresses added to the white list with ``addToContractWhiteList-ink`_` (as well as the contract owner) will be able to call this smart contract. If disabled, all addresses can call this smart contract.

### Permissions

- Address that deployed smart contract

### Parameters

- `contractAddress`: Address of the contract

- enable: Boolean that tells to either enable (if true) or disable (if false) the white list for that smart contract

### Managing Allow List for EVM Contracts

#### Description

A user will be able to call the smart contract only if their address is included in the contract allow list.

This short example uses web3 library and demonstrates how to add or remove a user address to/from the smart contract allow list. The contract address is stored in *myContractAddress* variable:

```
import Web3 from 'web3';
...
const helpers = new web3.eth.Contract(contractHelpersAbi as any,
  ↪ '0x842899ECF380553E8a4de75bF534cdf6fBF64049', {from: caller, ...GAS_ARGS});
await helpers.methods.toggleAllowed(myContractAddress, caller, true).send({from: owner});
```

#### Permissions

- Address that deployed smart contract

#### Parameters

- contractAddress: Address of the contract
- Address to add/remove
- enable: Boolean flag. True means address is included in the allow list and can call the contract. False means address cannot call the contract.

### addToContractWhiteList (Ink!)

#### Description

Add an address to smart contract white list.

#### Permissions

- Address that deployed smart contract

#### Parameters

- contractAddress: Address of the contract
- Address to add

### removeFromContractWhiteList (Ink!)

#### Description

Remove an address from smart contract white list.

#### Permissions

- Address that deployed smart contract

#### Parameters

- contractAddress: Address of the contract
- Address to remove

### 1.3.10 Governance-only Methods

The methods in this group can only be called by the root of the chain. They are not available for public use and are only listed for reference.

#### setChainLimits

##### Description

Sets some chain limits and starts enforcing them immediately.

- *collection\_numbers\_limit*: Total number of collections
- *account\_token\_ownership\_limit*: Total number of tokens that a single address can own
- *collections\_admins\_limit*: Total number of collection admins
- *custom\_data\_limit*: The maximum byte-size of token metadata
- *nft\_sponsor\_timeout*: The number of blocks between sponsored transfers for NFT tokens
- *fungible\_sponsor\_timeout*: The number of blocks between sponsored transfers for Fungible tokens
- *refungible\_sponsor\_timeout*: The number of blocks between sponsored transfers for Refungible tokens

##### Permissions

- Network Root

##### Parameters

- ChainLimits structure (see the description of parameters above)

## 1.4 Unity API

NFT Asset for Unity Framework aims to enable Unity developers to work with Unique Network blockchain without or with just a little knowledge about blockchain.

Unity Framework allows customising the behaviors of any gaming objects by means of writing scripts in C# and attaching them to any behavior. In order to make scripts reusable, they may be wrapped in a plugin library (e.g. Windows DLL), which can then be imported in any project. There are many plugins that are currently available for Unity developers. More information about Unity plugins may be found in Unity 3D documentation: <https://docs.unity3d.com/Manual/Plugins.html>

In order to use an imported plugin, Unity developer will use one of OOP principles called inheritance: The game object will extend the plugin, which means that gaming object will have some built-in features that come from plugin.

The Developer UI enables following actions:

- Create or import address with private key
- Display wallet balance in the network currency
- Create/Destroy collection
- Show list of collections owned by this wallet

Some features are yet to be tested/added:

- Add/Remove an admin for a collection
- Show list of collection admins

- Create/Transfer/Burn an NFT for a collection
- Set/View collection offchain data schema

For demonstration, please use [these instructions]([https://github.com/UniqueNetwork/nft\\_unity/blob/master/src/DemoApplication/readme.md](https://github.com/UniqueNetwork/nft_unity/blob/master/src/DemoApplication/readme.md))

More details coming soon...

## 1.5 .NET API

The .NET API allows Unique developers to connect their NFTs to .NET applications.

Details coming soon...

## 1.6 Wallet Integration Guide

This document is written for the wallet developers and intends to provide step by step guidance for integrating Unique and Kusama NFTs into the 3rd party wallets.

### 1.6.1 Unique

#### 1. User Collections

Step 1 is getting the list of collections, in which user owns tokens. There are two options to get these.

##### Option 1 - Traversing Events

This [PolkadotJS guide](#) explains how to tranverse events in a substrate based blockchain.

The events that we are looking for are *Transfer* in `transfer` extrinsic. It has parameters: Collection ID+Token ID, sender and recipient, which are the wallet addresses that exchanged NFT, and *ItemCreated* in `createItem (Mint)` extrinsic, which contains Collection ID and Recipient (wallet) address.

##### Option 2 - Manual Input

Sometimes traversing events may not be the most reliable or quick way to gather the full list of tokens for a user, so the wallet should allow manual user input for the collection by ID or name. In order to prepare for that input, the wallet application can read the full list of collections in the Unique network first. Collection IDs are sequential numbers that start from 1 and go up to the last created collection, which is:

```
api.query.nft.createdCollectionCount()
```

Each collection then can be queried with:

```
api.query.nft.collection(collectionId)
```

and will contain the Name and Description fields encoded as UTF-16, and TokenPrefix encoded as UTF-8 in response like this:

```

{
  Owner: 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY,
  Mode: {
    NFT: null
  },
  Access: Normal,
  DecimalPoints: 0,
  Name: [
    110,
    97,
    109,
    101,
    0
  ],
  Description: [
    100,
    101,
    115,
    99,
    114,
    105,
    112,
    116,
    105,
    111,
    110,
    0
  ],
  TokenPrefix: 0x70726566697800,
  MintMode: false,
  OffchainSchema: https://example.com/images/{id}.png,
  SchemaVersion: ImageURL,
  Sponsor: 5C4hrfjw9DjXZTzV3MwzrrAr9P1MJhSrvWGWqileSuyUpnhM,
  SponsorConfirmed: false,
  Limits: {
    AccountTokenOwnershipLimit: 10,000,000,
    SponsoredMintSize: 4,294,967,295,
    TokenLimit: 4,294,967,295,
    SponsorTimeout: 14,400
  },
  VariableOnChainSchema: ,
  ConstOnChainSchema:
}

```

Token prefix is used to display tokens in the wallet. The examples of refixes can be: BTC, ETH, etc.

### 2. User Tokens

Once the list of collections that a user (wallet address) has ever dealt with is ready, reading the list of tokens becomes a simple task. This query returns the list of user's tokens in one collection:

```
api.query.nft.addressToken(collectionId, address)
```

The return contains the list of token IDs. Return example:

```
[
  5243,
  6323,
  355,
  2888
]
```

### 3. Token Details

Token details allow the wallet to get access to token image and decode its metadata into a human readable format.

There are two types of token details: Common (or similarly structured) for all tokens in the collection, and details that are only relevant for one particular token (like a CryptoKitty name, for example).

Even though images and large metadata will be generally stored off-chain (due to cost and efficiency reasons), Unique enables 3rd party wallets to access this data without using any 3rd party APIs. The collection contains the *Offchain-Schema* field, which contains the schema string. Even though the collection owners can set an arbitrary string in this field, they are encouraged to use metadata standards in order to be compatible with Unique and 3rd party wallets. Currently there are two schema versions:

- ImageURL
- Unique

ImageURL is very limited. It only allows to set the URL template like “<https://example.com/images/{id}.png>”, which allows replacement of *{id}* placeholder in order to get the image for the token with a particular ID.

Unique schema is much more flexible. It allows not only to encode image URL templates, but also to set the URL for the API that stores token off-chain metadata, and define rules about what token on-chain data bytes mean and how to decode them into a human readable format. The [Data Schema](#) section describes how to do it. We encourage wallet developers to start implementation with ImageURL schema, then proceed to off-chain part of Unique schema, and finally implement the on-chain part of Unique schema.

Reading token on-chain data is done with a query that depends on the collection type. For NFT tokens, the *nftItemList* state variable should be used. For ReFungible collection it is *reFungibleItemList*.

For example, this query:

```
api.query.nft.nftItemList(collectionId, tokenId)
```

returns the NFT information:

```
{
  Owner: 5GrwvaEF5zXb26Fz9rcQpDWS57CtERHpNehXCPcNoHGKutQY,
  ConstData: <TOKEN METADATA>,
  VariableData: <TOKEN USER DATA>
}
```

The *ConstData* field contains the token metadata string that cannot be changed and is set when the token is minted.

The *VariableData* field (which, by the way, is also described by the schema) contains bytes that can be changed by the current owner, and usually will be changed by the application, but the wallet may allow users to change this (as long as the data stays within the schema).

## 1.6.2 Kusama NFT

TBD