

STAS 

Scope of the document

This is a technical documentation describing in detail the requirements to allow a licensee of the STAS opening and locking script - which is built on the native Bitcoin Script - from deploying the token solution in using the SDK and API from Taal.

The documentation herein is focusing on the deployed **stas-js library v0.2** which can be found on github: <https://github.com/TAAL-GmbH/stas-js>

Access to the repository is granted upon signing up over the Taal website under <https://www.taal.com/contact/>

Taal requires that a user intending on using the STAS tokenization SDK and API require to sign-up over the Taal console and signing of the STAS agreement.

The technical documentation will only outline the standard setup in using the SDK and API functionalities for the STAS script token solution and excludes the Atomic Swap functionality for now.

The documentation will be updated and made available to the community as we further develop and enhance the libraries of the SDK and API.

Audience

The intended documentation is tailored for anyone:

- with some knowledge of Bitcoin or blockchain
- Understanding JavaScript
- Using an SDK and integration of APIs

Document revision history

Version	Date	Author
Initial draft	05. December 2021	Simon Giselbrecht
Final version 1.0	14. January 2022	Simon Giselbrecht

Contents

1	Introduction.....	1
1.1	Introduction to Bitcoin “Smart” Contract	1
1.1.1	Variable data part:.....	3
1.1.2	The token mechanic’s part	4
1.1.3	Constant data	4
1.1.4	Transaction structure	7
1.1.5	Cost of transactions	8
1.2	What is MAPI / TAPI.....	9
1.2.1	MAPI	10
1.2.2	TAPI.....	10
1.2.3	TAPI Direct	11
1.3	What is the Taal Console	11
1.4	What is Watsonchain.....	11
2	Getting started with TAPI / MAPI	12
2.1	Taal Console.....	12
2.2	Utilizing MAPI and Taal API (TAPI)	13
2.2.1	MAPI	13
2.2.2	TAPI.....	13
3	Getting started with the SDK and API.....	14
3.1	Setup.....	14
3.1.1	Using in browser	16
3.1.2	Using Node.js.....	16
4	How to use the SDK	17
4.1	Flow of STAS tokenization.....	17
4.2	Transaction fees:.....	21
4.3	Environmental Variables.....	21
4.4	Utilization of each core function of the STAS JS SDK	22
4.4.1	Contract	22
4.4.2	Issuance	24
4.4.3	Transfer	25
4.4.4	Split	26

4.4.5	Merge	27
4.4.6	Atomic swap	29
4.4.7	Redemption	29
5	How to use the API	32
5.1	What API functionalities do we have.....	32
5.1.1	Get all tokens.....	32
5.1.2	Get token by Id	32
5.1.3	Get address token balance	33
5.1.4	Get script transactions.....	33
5.1.5	Get token transactions	34
5.1.6	Get token output details.....	34
6	Performing Testing	35
7	Appendix.....	36
7.1	Appendix A: Standard Metadata types	36

1 Introduction

Substantiated Tokens from Actualized Satoshis (STAS) is a token protocol owned by TAAL that turns regular UTXOs (Unspent Transaction Output , unspent coins basically) into P2TAS UTXOs (unspent tokens) via the usage of Bitcoin OP scripts. You can create the STAS token from a satoshi or any amount of satoshis (considering dust limit on the network), which is the smallest division of a bitcoin and the base unit of exchange in the Bitcoin Satoshi Vision (BSV) network. There are 100,000,000 satoshis in 1 bitcoin.

The STAS locking and unlocking scripts ensure that a set of token rules are propagated from the issuance transaction till the redemption transaction.

Further STAS utilizes a set of tools that are provided by Taal to ease the usage of developing and deploying of STAS token solution.

For a detailed overview over all of the services that are offered in addition to the ones requires to run STAS successfully please refer to the following diagram

https://www.taal.com/wp-content/uploads/2021/10/TAAL_Infrastructure_Diagram.pdf

In the subsequent sections we will provide an overview of the toolsets that should be used when using STAS as a standard token solution and linked toolsets.

1.1 Introduction to Bitcoin “Smart” Contract

BSV network supports a wide range of smart contracts that are based on using the scripting language known as Bitcoin script. Many concepts of tokenization or issuance of smart contracts follow a common logic where tokens that are issued effectively create new token units on layers on top of BSV and require different layers of logic to be run on top of it (see appendix for examples of token architectures).

One of the most powerful inventions on the BSV blockchain was the introduction of the concept for the usability of OP_PUSH_TX that allows deployment of smart contracts. An OP_PUSH_TX contract in a locking script is divided into code and data allowing it to become readable on-chain through a blockchain explorer such as whatsonchain.com. OP_PUSH_TX allows inspection of the entire transaction inside a contract, including all inputs and outputs. This opens boundless possibilities for all kinds of smart contracts on Bitcoin and can be built in using any token Layer architecture determined suitable for the use case. This is the case for Layer 1 and Layer 2 solutions where the token logic is represented off-chain by running proprietary code on proprietary servers, and thus a certain amount of trust needs to be assigned to the platform itself. On the other hand, STAS script has no layer protocol, as it is not represented as a layer. It is an optional standard using only Bitcoin native script, much like native Bitcoin smart contracts. It is a tokenization standard that uses the smart contract capabilities of Bitcoin itself.

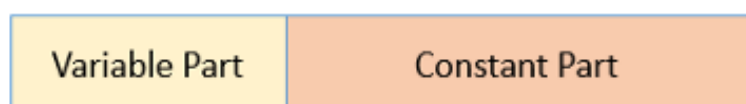
Structure of a contract

A Bitcoin smart contract has various components that must play together in order to allow a token to be minted, sent / received and redeemed. A detailed explanation of the Bitcoin Smart Contract was given in the STAS white paper.

On a technical level, P2PKH scripts establish the specific requirements that, in order to spend bitcoin, a user must provide an ECDSA signature that matches the public key whose hash is specified by the script.

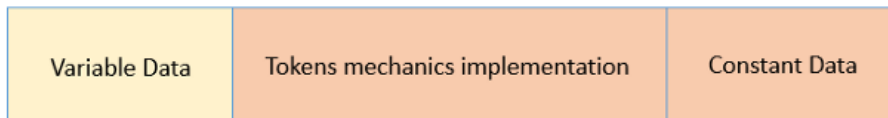
The token contract consists of two main parts that are held as smart transaction outputs and can be in:

- Single Satoshi — representing single token (e.g. utility token: event, cinema, bus ticket etc.), meaning it's indivisible and can't be split during its life cycle
- Multiple Satoshis — representing an envelope with multiple tokens (e.g. USD-pegged tokens), let's call it an *envelope* type



₿ STAS

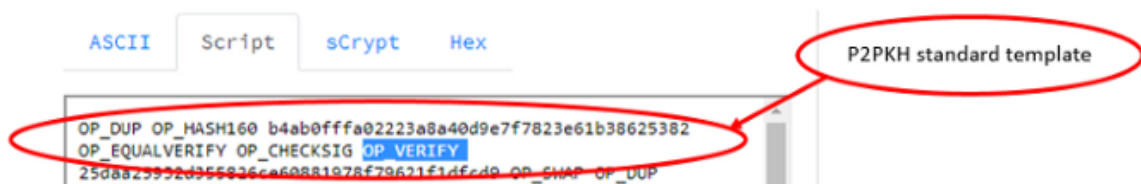
- The variable part is identical to regular Bitcoin's payment template that passes on possession of the tokens through ECDSA usage.
- The constant part can neither be changed nor omitted through the time of the token's life.
- Each Satoshi now represents a token and stops its regular functioning, although it can represent something cheaper or more expensive than one Satoshis itself. To illustrate this based on an example means that Alice can't move the tokens to Bob unless she keeps this exact template as a locking script in sending a transaction output, all she can change is the possession address in the variable part.
- If we look closer there are three main parts that play a key role in how to use STAS.



1.1.1 Variable data part:

This part is intentionally at the very beginning of the script and is just a regular P2PKH template, the reason for this is to allow a maximum possible compatibility for existing wallets and browsers, providing their searches by specific raw ECDSA addresses wrapped with this fixed template.

As shown below this shows the designated part of the current Satoshi holder.



***Note:** For maximum compatibility, in order to keep the P2PKH template unaltered OP_CHECKSIG wasn't exchanged for OP_CHECKSIGVERIFY, instead the OP_VERIFY opcode was added separately.

1.1.2 The token mechanic's part

This is the *code* part of the smart contract that ensures:

Token spending is possible only if the next UTXO has the same locking-script as the UTXO being spent has, apart from new possessor update

Spending by transaction with UTXO carrying regular Bitcoin locking scripts (*P2PKH*, *P2PK* etc.) fails, unless it is a *P2PKH** type carrying the redemption address set in the contract upon its issuance. Meaning, only the issuer has an ability to turn an asset representing Satoshis/Bitcoins back into their original regular use as native tokens (by releasing the underlying assets, e.g., USD, Gold etc.).

The amount of Satoshis each representing a token upon spending must be one of the following:

- unaltered — in case of plain transfer (either an envelope or a single-token type)
- split in two — in case of spending a transaction having two outputs (e.g., partial spending with change in second output) each with locking scripts identical to the one being spent (apart from possessor part)

What constitutes the state pushed through in these *stateful* TXs is the possession address updated each hop AND (only in case of envelope-type with multiple tokens) the number of tokens (that may be reduced through splitting).

1.1.3 Constant data

The token protocol ID is a field by which the token types will be identified in our case this represents the STAS protocol and the TXID corresponds to a pointer to a special transaction, which contains:

- Initial issuance contract (e.g., between issuing entity and the client) with all its legal terms, conditions, license and any other possible necessary information related to this token/s attributes and its issuance. These details can be open for anybody to read, or fully / partially encrypted depending on the use case. The restriction if it is encrypted means though that only the addresses that have the unlocking script can read the details thereof.

- Exact satoshis balance predetermined to be transformed into tokens in the subsequent transaction (which will be the issuance transaction) with the so called smart token output(s).

Let's call it — a spendable data-TX (or a spendable OP_RETURN TX), since it carries data in the way regular OP_RETURN data-TXs do, but with the difference that it has money in it and thus may be

For an example we build a spendable data-transaction (in script language an OP_RETURN transaction), that contains data in the way a regular data transaction has, with the main difference being that it has money in it and can be spent.

```

OP_PICK OP_CAT OP_ELSE 346650137 OP_SWAP OP_CAT -11400 OP_CAT
OP_ENDIF OP_CAT 8 OP_PICK OP_IF 9 OP_PICK 8 OP_NUM2BIN OP_CAT 5
OP_PICK 1354102 OP_CAT 9 OP_PICK OP_CAT 5 OP_PICK OP_CAT OP_CAT
3 OP_ROLL OP_ROT 9 OP_PICK OP_ADD OP_NUMEQUALVERIFY OP_ELSE 3
OP_ROLL OP_ROT OP_NUMEQUALVERIFY OP_ENDIF 4 OP_PICK OP_IF 5
OP_PICK 8 OP_NUM2BIN OP_CAT 346650137 5 OP_PICK OP_CAT -11400
OP_CAT OP_CAT OP_ENDIF OP_HASH256 OP_EQUAL OP_RETURN
27ed5442ab151b32db7b586d433879d84af3398a
723c903d83cdce9f32246a451652f4b4f469ef78d630bf0f22770cb16bbc187
b
  
```

Total Output: 117,432 sat

The reason for these fields to be included after the OP_RETURN opcode is to allow compatibility for apps/browsers performing their searches on OP_RETURN transactions (unspendable data transactions), where OP_RETURN opcode is followed by the protocol ID and the rest of the data, as in this specification.

It is best to demonstrate the data transaction (pointed out by the second field) with a self-explanatory example as show below.

Scriphtash: 1,000 sat ↕
[66b0d03ce29ff7fec7cf2ed5eb3a88fa80e](#)
[39e4e79eb0c98ac9c6fd260d5cea0](#)
 type: nonstandard

ASCII **Script** sCrypt Hex

```
v0%0Y20U0l0000yb0000~jM0 This is a token issuance
contract between SBI Bank and Alice. This TX provided to
Alice, signed with ANYONECANPAY flag by SBI Bank, has to be
signed and published by Alice to Bitcoin's global ledger.
Hereinafter, 10 USD funds will be transferred to SBI Bank,
branch number: XXX, account number: XXXXXXXX. As a result
the bank is obligated to issue 1000 tokens on behalf of
Alice, within two working days, to raw address:
2f2ec98dfa6429a028536a6c9451f702daa3a333 provided by Alice.
Each of the 1000 satoshis in the output of this transaction
will be transformed into a token representing 1 USD cent.
Serial issuance number: XX-XXXX-XXXXXXXXXX. Any partial
amount of these tokens may be redeemed by sending them to
Bitcoin raw address:
25daa25932d355826ce60881978f79621f1dfcd9, which is
officially identified with SBI bank and will be set in
tokens smart-contract outputs in a hardcoded immutable
manner. Created by Stas Trock! Test 0!
```

Pre-allocated satoshis for transformation into token meaning in next TX by the issuer

The legally accountable contract between the issuer and the client

This is how the same spendable data transaction looks as *Script* (ASM):

Scriphtash: 1,000 sat ↕
[66b0d03ce29ff7fec7cf2ed5eb3a88fa80e](#)
[39e4e79eb0c98ac9c6fd260d5cea0](#)
 type: nonstandard

ASCII **Script** sCrypt Hex

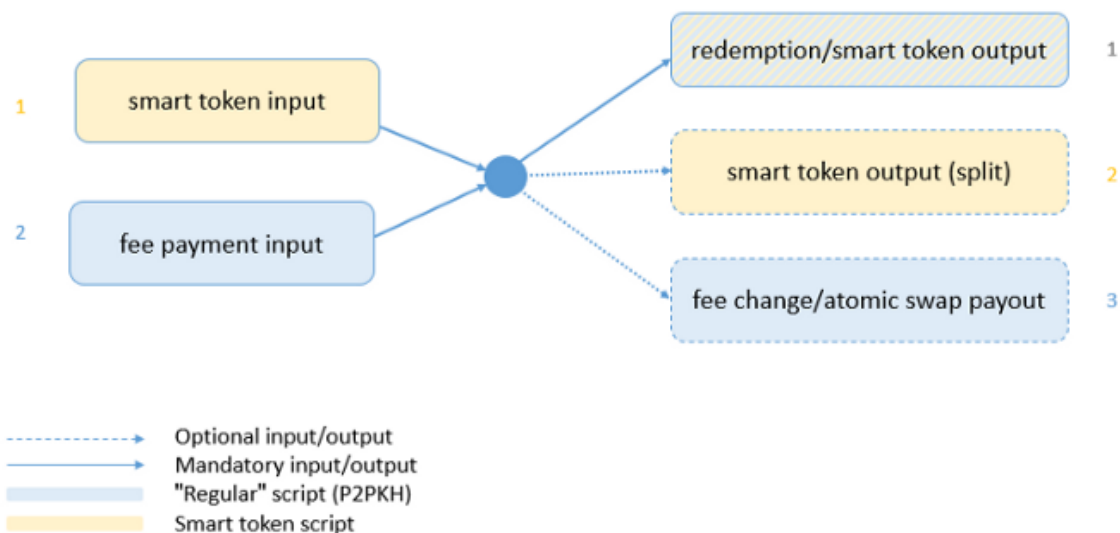
```
OP_DUP OP_HASH160 25daa25932d355826ce60881978f79621f1dfcd9
OP_EQUALVERIFY OP_CHECKSIG OP_RETURN
2020202054686973206973206120746f6b656e2069737375616e636520636f6
e7472616374206265747765656e205342492042616e6b20616e6420416c6963
652e0a20202020546869732054582070726f766964656420746f20416c69636
52c207369676e6564207769746820414e594f4e4543414e50415920666c6167
206279205342492042616e6b2c2068617320746f206265207369676e6564206
```

The TXID of this transaction is the immutably set pointer in tokens' smart output for the rest of their life and every subgroup of tokens split from the initial group will forever carry the same pointer.

1.1.4 Transaction structure

Various tokens will require and consist of different issuance degrees and complexities. Whilst some cases will include multiple legal documents others will be very simple or even completely automated.

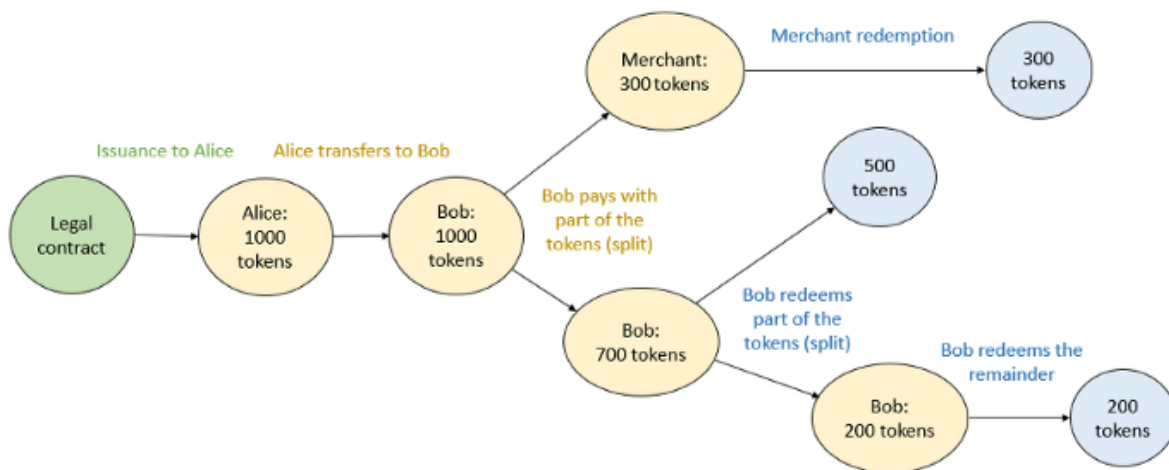
To demonstrate this, we have put together in forming a smart token contract. The outputs may either be used for redemption using the regular P2PKH outputs or for transfer / split a smart output. Further it could be using the optional smart output (when used in the case of a split only), or a “regular output that is used either for fee payers change or in the case of n atomic swap, the token seller’s payment reception.



To demonstrate this as part of a token issuance, this could work in the following way:

- Alice requests a legal authorized entity to issue on her behalf 1000 tokens (providing it with her address for reception).
- The entity creates an agreement (fills up the template) in the form of spendable data TX with pre-allocated amount of satoshis in it (1000 satoshis) corresponding to the requested number of tokens.
- The terms/conditions of the agreement state Alice’s required actions (e.g., transfer corresponding amount of fiat money to the entity) and are signed by the entity part (with ALL | ANYONECANPAY flags, to allow Alice only to add her signature).

- Once Alice signs and publishes the agreement to the global ledger and performs the stated required actions, the entity issues the tokens on her behalf (with her raw address).
- Alice can now use/spend/transfer all at once or part of the tokens at her will, and the subsequent possessors may redeem the tokens at their will (*permission-less* type).



1.1.5 Cost of transactions

The goal and misconception of miners is that they compete for including various transactions in blocks by giving lower fees than the others. Compared to other miners, Taal has the ability and is capable of processing low dust, 0 fee, and provides customized fees that can be discussed and agreed upon with our commercial team, which however does require to register on the Taal console.

Moreover, not all transactions will require to have “instant” confirmed (mining not in the next block, but at any point later on) fall today in that range of low dust, zero fee or a customized fee right from the start.

Since currently fee payment depends mainly on the transaction size, this script implementation is space-optimized, rather than time (in the future it may be adapted to other miners’ policies).

It went through thorough optimization, considering every *opcode*, so that any further non-negligible size reduction to include this logic is not expected to be feasible.

Thus, defining a clear cost over the tokens cost related to transaction fees cannot be completed or stated generally. For the purpose of allowing the calculation over a transaction, a user of STAS should use the following calculations as a guideline by pulling this into their own excel to perform calculations over transaction costs.

	Input fields in yellow	Explanation
Sats/BSV	100000000	Number of Sats in 1 BSV, dont change this, its the constant
USD/BSV		Input current USD/BSV price from coinmarketcap
Sats/kB		Input expected transaction fees in Sats
Tx size kB		Input the expected transaction sizes to be submitted
Tx fee BSV	$= (C5 * C6) / C3$	Calculated
Tx fee USD	$= C7 * C4$	Calculated
Number of tx		Input the expected number of transactions
Total	$= C10 * C8$	Calculated what the total price is of submitting a transaction

1.2 What is MAPI / TAPI

Merchant API (MAPI) is a product that was developed by nchain and is publicly available for anybody to use. However please note the following:

- A client can choose to utilize any of the three options to submit transactions or tokens in using the toolsets explained here in, but it is strongly recommended to use the standard implementation route in using TAPI or TAPI Direct. This is mainly due to the following reasons:
 - A client can choose not to register over the taal console and use their own API keys and urls to call and use the MAPI standards as documented and publicly available, but Taal cannot provide and support on the implementation and subsequent integration to the client application.
 - A client can choose to sign-up and receive a standard (500sats/ kb), lower fee (250sats/kb) or a custom fee API key that can be used with TAPI or TAPI Direct. The benefit here is clear that it's a Taal product and we do support clients if they encounter issues in using the tooling.

1.2.1 MAPI

Merchant API (mAPI) is an additional service that enables merchants to get policy and fee quotes for submitting transactions, submit the transaction and query the transaction status and get a fee quote from miners.

The API enables merchants accepting Bitcoin transactions to get all of the assurance they need to be able to accept transactions on a 'zero confirmation' basis (see Confirmation).

Merchants can find out in advance (even before broadcasting a transaction) what transaction fee is required by Miners to mine it.

The usage of MAPI allows anybody to ask for fee quotes from miners such as Taal, Matterpool or other miners. For Taal specific infrastructure usage Taal does provide a direct access in using TAPI and TAPI Direct services.

1.2.2 TAPI

Taal offers clients the possibility to use allows customers in using Taal API (TAPI) that is a modified version of MAPI that will allow to send transactions, using their API key (containing key package parameters) to TAAL's nodes directly and get call backs on transaction status to see if they have been validated or mined.

In using TAPI clients can choose to use the standard, lower, no fee or custom defined sats / kb fee keys when sending a transaction's through Taals transaction processing capabilities. Whilst the validation of a transaction is picked up by the other mining nodes can be viewed through end points of the WOC API.

1.2.3 TAPI Direct

TAPI Direct functions like the TAAL MAPI path. To minimize complexity, TAPI Direct essentially allows a customer to submit a raw transaction to TAAL's endpoint to be passed on direct to the node. TAPI Direct does not provide any pre-node validation or any of the callback notifications.

1.3 What is the Taal Console

The Taal console allows customers to track their activity per API key, check their account details, send messages to TAAL support, can access TAAL Services Platform to purchase additional services and / or adjust current services they have deployed.

1.4 What is Whatsonchain

Whatsonchain is the first and most comprehensive BSV blockchain explorer that can be utilized by developers as it provides more succinct data in an easy-to-use package that allows the searching over any piece of information, such as a transaction ID or a crypto wallet address, by providing all the details of when and who mined the data related to that transaction. It also gives a representation of who is mining what blocks and how large those blocks are.

2 Getting started with TAPI / MAPI

Registering over the Taal console will allow a user to obtain the test net API key, that is required to build, send and submit transactions in using our standard services over TAPI / MAPI.

2.1 Taal Console

Use the link: <http://console.taal.com/> to register as an account for your company / user wanting to use our services from TAPI / MAPI for building and submitting transactions with or without Taal services.

The focus here in lies on using TAPI for simple integration to read and write transactions, using the Taal console API

Once registered the API keys are generated for you and shared with you via email or by our sales team.

Taal provides a set of standard API keys to submit transactions which are then validated, and then ensure that these are mined in the next block depending on the transaction fees submitted:

1. If you require a transaction from being validated by a node and then mined, it is recommended to submit transaction in using the standard 500 sats / kb transaction,
2. If you do not require a transaction from being validated by a node and instantly processed by miners, a transaction could be submitted with 250 sats / kb,
3. There is also the ability to obtain lower rates, which however can only be granted upon submitting a direct request to Taal. Further this also means that transactions may be validated by a node, but the mining will be delayed, as miners do not pick up low dust, or low fee transactions.

The API key can then be used for the application and called on to use when building and submitting transactions through TAPI.

2.2 Utilizing MAPI and Taal API (TAPI)

2.2.1 MAPI

For a more comprehensive view over the usage of MAPI please follow the existing documentation which is available here: <https://developers.taal.com/#merchant-api-mapi>.

2.2.2 TAPI

TAPI is a Taal specific endpoint which allows the submission of transactions directly to Taal's transaction processors and does not provide any of MAPI's extra functionalities e.g. providing Merkel proofs, callbacks, fee quotes etc.

We recommend that a client using the STAS SDK library functions to create contracts and issue (mint) tokens submit the transactions through TAPI at the correct fee rate so they are relayed to the rest of the network and can then be picked up by the STAS API.

3 Getting started with the SDK and API

A Software Development Kit (SDK) is available for both Javascript and Golang. The Javascript SDK works on both NodeJS and web browser. The focus for now is on the Javascript version of the SDK, as this is the language of the SDK most used to-date for STAS.

The main purpose of the SDK is to compose the following OP scripts for STAS transactions:

- Locking scripts: These scripts are the address to which the funds are sent.
- Unlocking scripts: These scripts are added to the locking scripts to spend the funds.

A testing environment is available in which you can mine the STAS UTXOS with the token owners address. To view the token details you would look up the hashed redemption address and symbol.

Depending on the use case some or all functionalities would / could be utilized for deployment of the SDK functionalities.

- The library provides example token lifecycles for the following use cases:
- NFT tokens
- No fee transactions for tokens
- Normal token life cycle example
- Large token issuance
- Large token split

The library further contains a list of tests that can be used on test net or mainnet. For details to testing please refer to that specific section.

3.1 Setup

To get started a project using the SDK, must be in possession of a main net key which can be obtained from registering in the Taal Console as outlined in the steps above.

Cloning of the SDK from [git clone https://github.com/TAAL-GmbH/stas-js.git](https://github.com/TAAL-GmbH/stas-js.git)

The term UTXO refers to the amount of digital currency someone has left remaining after executing a cryptocurrency transaction such as bitcoin. The letters stand for unspent transaction output. Each bitcoin transaction begins with coins used to balance the ledger. UTXOs are processed continuously and are responsible for beginning and ending each transaction. Although confirmation of transaction results in the removal of spent coins from the UTXO database, a record of the spent coins still exists on the ledger.

A UTXO database is used to store change from cryptocurrency transactions. This database or ledger is initially set to empty or zero. As transactions multiply, the database becomes populated with change records from various transactions. When a transaction is completed and there are outputs that aren't spent, they are deposited back into a database as inputs that can be used at a later date for a new transaction. Cryptocurrency transactions—are similar to cashier checks. You cannot exchange them for custom amounts and must spend the entire amount stored in that data byte.

A detailed overview in how the UTXO concept is important is explained in the contract functionality.

For Contract & Funding UTXO's (merge UTXO is different) the following format should be used as explained below.

Function Returns: All functions return a serialised transaction (a hex) that can then be broadcast to the blockchain.

You can implement the following functions to ensure that the STAS SDK works on the Bitcoin network by using our lifecycle test examples in how to interact with the SDK:

- **getTransaction:** This function provides the unspent transaction output (transaction ID and vin) required to spend the funds. You can search by using the following:
 - the recipient public key hash and `TOKEN_ID`
 - `TOKEN_ID` for all the unspent outputs (defacto tokens in circulation)
- **getFundsFromFaucet:** Is not technically required, but this function is the source of satoshis, which is used to start the STAS token life cycle and pay the transaction fees. A faucet is an online service that dispenses Bitcoins for free on taalnet or testnet. The faucet does not exist on mainnet.

- **Broadcast:** This function sends transactions to a full node. The transaction ID and vin are captured in the creation of the transaction. This function should return once a Bitcoin node has accepted the transaction into its mempool. After the transaction has been accepted, it will take time for the transaction to propagate to other nodes and blockchain explorers.

3.1.1 Using in browser

This library can be used within the browser, but only when configured correctly. `stas-js` is dependent on `bsv.js` which in turn is dependent on `BN.js`. `BN.js` assumes that `Buffer` exists globally as it does in NodeJS, however, this is not the case in the browser.

The `bsv.js` is available as a standard JS library and it includes a copy of `safe-buffer`, which comes to the rescue. By adding this to our `index.html`, the `Buffer` polyfill is loaded for us.

3.1.2 Using Node.js

Download the library and import to `node.js` and deploy the `launch.js` file.

4 How to use the SDK

The SDK library contains a set of functionalities that will build the required transactions that form part of a token life cycle. In this version of the token life cycle documentation we will only focus on the generic use case for tokenization in using the STAS scripts.

The SDK functionalities wrap around the STAS script, which allows a UTXO set to be transformed from Satoshi's to STAS tokens, sent / received, traded and redeemed. Please note that the Swapping functionalities are not included as part of this first documentation, as it will only be rolled out in Q1 2022.

4.1 Flow of STAS tokenization

A detailed view over the STAS token life cycle can be found in the white paper but is herein included for simpler reference purposes only.

To use the STAS script, Taal has developed an SDK currently in Go-Lang and JavaScript form, that allows the functionalities of the STAS script to be integrated, which can be then be utilized to build a transaction to issue, transfer, trade or redeem a token that is using the STAS locking and unlocking script.

The STAS token lifecycle includes the ability to build any type of application on top using the locking and unlocking script. Depending on the use case all or only selected functions can be utilized as required by using the SDKs basic functionalities.



Figure 1: STAS Token Lifecycle

A typical lifecycle to issue a STAS token consists of the creation, issuance, transfer, and redemption of a token. However, there are multitudes of use cases that may require the ability to also merge and split tokens as a whole or in a fractionalized form. The above Figure 1 illustrates an example life cycle for any type of token issuance.

1. The client defines the token scheme for the terms and conditions (tokenomics) of the STAS tokens to be issued.
2. Ensure that the client wallets contain the spendable Satoshis (Sats).
3. The client builds a contract transaction that contains the terms and conditions scheme (tokenomics), which includes the amount of Sats. The scheme will lock and transform Sats during the Minting process.
4. The client signs the contract and subsequent transactions with a private key, which are submitted to the block chain.
5. Once submitted to the blockchain, the STAS locking script reserves the Sats and transforms it into spendable minted (issued) STAS tokens.
6. Once the minting is completed, the STAS tokens becomes tradeable using the STAS locking or unlocking scripts functions.
7. The redemption takes place once a STAS token is destroyed and the underpinning amount of sats is sent back to the issuer address
8. The redeemed STAS token transforms into an unspent native Satoshi, which can be minted again under a new contract transaction.

Table 1: STAS functionality

Functionality	Description
STAS token scheme	<p>Defines in a JSON schema for the Minting terms, which enables the API to extract information about the STAS token, create business logic, reward structure, data validations, and so on.</p> <p>This must include Name, TokenId, Description of the token, Issuer</p>

Functionality	Description
	name, Country, Legal form, Email address, Website, Governing law, Terms of use, Icon, and Ticket symbol.
Contract transaction to BSV network	Parses the defined Minting token JSON scheme into a legal text document, which is published in the BSV blockchain network, and enables you to view the token terms, company details, and so on.
Issuance of STAS	<p>The issuance blocks the Satoshis and mint STAS token with the defined value from the Token schemes.</p> <p>The Sats (input) are converted to STAS tokens/coins (output), which are sent (unspent) to 1 (Transferred) or 1:n (Split) to various BSV addresses using the BSV accounting method (called UTXO).</p> <p>Any token can be Split into 1:n, except an NFT. The NFT tokens cannot be split or merged, and data can only be added as part of the issuance process. However, an NFT token can be transferred, swapped, or redeemed.</p>
Swap	The atomic swap functionality is a trustless exchange of different assets without counterparty risk within the BSV eco-system. It enables a swap between any chosen token/coin against each other in using the same underlying BSV native coins within the network.
Split of STAS 1:n	When you send STAS to more than 1 address, it triggers the splitting of the STAS, which can be in whole or fractional amounts to multiple recipients.
Transfer of STAS 1:1	In a STAS transfer transaction, the STAS are sent from a different address (spent) to a new address (becoming an unspent) STAS, which can be used to create new transactions (outputs).

Functionality	Description
Merge of STAS	The merge functionality enables you to send (input) and split multiple STAS tokens to one address (output). It allows to merge 2 inputs together at the same time.
Redemption	The holder of a STAS token can redeem the STAS to the BSV native coins by sending back the STAS to the original address.
Mergesplit of STAS	The merge split functionality is required, if a STAS token address wants to merge two STAS UTXOS (from the same owner) and split these up into 2 different UTXOs that are sent to different addresses or owners
Redemption split	The redemption split function enables you to redeem one subset of a STAS UTXO while the rest are split and sent to (up to maximum 3) other different addresses.
API	An API and associated key enable you to broadcast the tokens/coins across the BSV network by using the transaction services. As well as and retrieve information regarding token balances, token details etc.

4.2 Transaction fees:

The mining fee is set in the config.js file. The default is currently 500 sats per 1000 bytes which is 0.5 sats per byte. Change the sats to whatever you expect to pay.

4.3 Environmental Variables

The following environment variables must be set inside the .env file
 API_USERNAME= The API username
 API_PASSWORD= The API password
 NETWORK= The network that the tests will run on. 'livenet' is the main network

4.4 Utilization of each core function of the STAS JS SDK

4.4.1 Contract

The contract function creates a transaction that contains an OP_RETURN ¹ and UTXO ² that contains a JSON Schema (also known as the tokenomics of a token) defines a token. The contract function takes the contract and funding UTXO, creates a new transaction and signs it using their relevant private keys. The value of the token is defined by the Supply and Sats Per Token. The Supply is the number of Satoshis to be converted to Tokens and the Sats Per Token is the value of a single token in satoshis. The supply amount must be equal or less to the satoshi value of the contract UTXO supplied.

Fees are an important aspect for creating and sending a contract and issuance transaction in using STAS. As explained earlier funding of transactions is an important aspect which are required to ensure a minting of STAS tokens to occur properly.

The contract and transactions fees require the possession of one or two addresses with a UTXO set that will a) fund the contract transaction and b) pay for transaction fees.


UTXO's are the unspent output from a transaction. When it is used as in the input of another transaction it is then spent and no longer a UTXO. For details also see the STAS white paper for a proper explanation of UTXO.

The contract function takes the following arguments:

9. Issuer Private Key - The private key for the funding of the contract UTXO
10. Contract UTXO - The funds in UTXO format for the contract (the amount of satoshis tied up in the contract)
11. Payment UTXO - The funds in UTXO format for the transaction fee

¹ OP_RETURN - a script opcode that marks the output as invalid (unspendable, ie does not contain bitcoin). It is used to store arbitrary data that can be stored on the blockchain.

² UTXO (Unspent Transaction Output) - The output of a transaction that has not been spent (consumed) in another transaction.

12. Payment Private Key - The private key for the payment of transaction fees
13. Schema - The schema that defines the tokens, it has the following properties;
14. token ID - the public key hash of the issuer private key(see  [Public Key Hash](#) for details)
 - Symbol - The type of token
 - Supply - The number of Satoshis to be converted to Tokens
 - SatsPerToken - The number of Satoshis per token
 - TokenSatoshis - The number of Tokens supplied

Line 54 below shows the implementation of the contract function with the required arguments defined lines 1 -52

Line 62 below shows the transaction broadcast to the blockchain

```
1  const contractHex = contract(  
2    issuerPrivateKey, // the private key of the issuer  
3    contractUtxos, // the utxo containing the sats for the contract  
4    fundingUtxos, // the utxo containing the sats for the fees  
5    fundingPrivateKey, // the private key for the fees  
6    schema, // the schema defining the contract and token  
7    supply // the total number of sats for the contract  
8  )
```

4.4.2 Issuance

The issue function spends the outputs from the contract and creates one or more token outputs (P2STAS). This function creates the token itself. The token ID can be retrieved and token details viewed after this function has completed successfully. The number of tokens, their destination and optional data is set by the `issueInfo` parameter. The contract UTXO contains the contract funds (ie the amount of satoshis the tokens are worth) and the payment UTXO contains the transaction fees. As with contract, the issue function creates a transaction and signs it with the relevant private keys. The function takes a boolean argument, `isSplittable` which defines if the tokens generated from the contract can be split into further parts. There is also a version parameter to define the version of STAS script used (currently only version 2 is available).

Issue Function Arguments;

1. Issuer Private Key - The private key for the funding of the contract UTXO
2. IssueInfo - An array containing the addresses to issue the tokens to, the amount of tokens and optional data
3. Contract UTXO - The funds in UTXO format for the contract (the amount of satoshis tied up in the contract)
4. Payment UTXO - The funds in UTXO format for the transaction fee
5. Payment Private Key - The private key for the payment of transaction fees
6. `isSplittable` - sets a flag to determine if the tokens can be split
7. Symbol - The token symbol set in the contract
8. Version - The STAS script version, currently only version 2

On lines 1 - 11 we can see that the total amount of tokens (10000) are split into two, 7000 to Alice and 3000 to Bob.

```
1  const issueInfo = [ // defines the addresses and amount of tokens to be sent as well as optional data
2    {
3      addr: aliceAddr,
4      satoshis: 7000,
5      data: 'one'
6    },
7    {
8      addr: bobAddr,
9      satoshis: 3000,
10     data: 'two'
11   }
12 ]
13  const issueHex = issue(
14    issuerPrivateKey, // the owner of the contract transaction
15    issueInfo,
16    {
17      txid: contractTxid, // the contract UTXO, this contains the satoshi value for the tokens
18      vout: 0,
19      scriptPubKey: contractTx.vout[0].scriptPubKey.hex,
20      amount: contractTx.vout[0].value
21    },
22    {
23      txid: contractTxid, // The funding UTXO for fees
24      vout: 1,
25      scriptPubKey: contractTx.vout[1].scriptPubKey.hex,
26      amount: contractTx.vout[1].value
27    },
28    fundingPrivateKey,
29    true, // isSplittable
30    "TEST-TOKEN",
31    2 // STAS version
32  )
33  issueTxid = await broadcast(issueHex)
```

4.4.3 Transfer

The transfer function allows a user to take an amount of tokens and send them to another address. The function takes a STAS UTXO and assigns it to another address. This is a 1:1 transfer.

Transfer function arguments;

1. STAS Token Owner Private Key - The private key of the owner of the STAS tokens to be transferred
2. Contract Public Key - The public key of the issuer of the contract // NOT REQUIRED
3. STAS UTXO - The UTXO containing the STAS tokens to be transferred
4. Destination Address - The address the tokens are to be transferred to
5. Payment UTXO - The funds in UTXO format for the transaction fee

6. Payment Private Key - The private key for the payment of transaction fees

```
1  const transferHex = transfer(  
2    bobPrivateKey, // the owner of the tokens to be transferred  
3    {  
4      txid: issueTxid, //The STAS UTXO containing the STAS tokens to be transferred  
5      vout: 1,  
6      scriptPubKey: issueTx.vout[1].scriptPubKey.hex,  
7      amount: issueTx.vout[1].value  
8    },  
9    aliceAddr, // the address of the tokens recipient  
10   {  
11     txid: issueTxid, // The funding UTXO for fees  
12     vout: issueOutFundingVout,  
13     scriptPubKey: issueTx.vout[issueOutFundingVout].scriptPubKey.hex,  
14     amount: issueTx.vout[issueOutFundingVout].value  
15   },  
16   fundingPrivateKey  
17 )  
18 transferTxid = await broadcast(transferHex)  
19
```

4.4.4 Split

The split function takes an existing STAS UTXO, splits it and assigns it up to 4 other addresses. This is useful when a user would like to take a number of tokens, divide them and send them to up to 4 addresses.

Split function arguments;

1. STAS Token Owner Private Key - The private key of the owner of the STAS tokens to be transferred
2. Contract Public Key - The public key of the issuer of the contract -- NOT REQUIRED
3. STAS UTXO - The UTXO containing the STAS tokens to be transferred
4. Split Destination Addresses - An array of addresses and token amounts the tokens are to be transferred to
5. Payment UTXO - The funds in UTXO format for the transaction fee
6. Payment Private Key - The private key for the payment of transaction fees

```
1 // Split tokens into 2 - both payable to Bob...
2 const bobAmount1 = transferTx.vout[0].value / 2 // the utxo from the previous transaction
3 const bobAmount2 = transferTx.vout[0].value - bobAmount1
4 const splitDestinations = [] // an array containing destination addresses and token amounts
5 splitDestinations[0] = { address: bobAddr, amount: bobAmount1 }
6 splitDestinations[1] = { address: bobAddr, amount: bobAmount2 }
7
8 const splitHex = split(
9   alicePrivateKey, // the owner of the tokens to be split
10  {
11    txid: transferTxid, //The STAS UTXO containing the STAS tokens to be Split and transferred
12    vout: 0,
13    scriptPubKey: transferTx.vout[0].scriptPubKey.hex,
14    amount: transferTx.vout[0].value
15  },
16  splitDestinations, // The array of addresses and amounts
17  {
18    txid: transferTxid, // The funding UTXO for fees
19    vout: 1,
20    scriptPubKey: transferTx.vout[1].scriptPubKey.hex,
21    amount: transferTx.vout[1].value
22  },
23  fundingPrivateKey
24 )
25 splitTxid = await broadcast(splitHex)
```

4.4.5 Merge

The merge function is used when you have two existing STAS UTXO's (outputs from other transactions containing STAS tokens) from the same owner and wish to merge them into a single UTXO and assign to an address. The tokens to be merged must have the same private key (same owner)

Merge function arguments;

1. STAS Token Owner Private Key - The private key of the owner of the STAS tokens to be merged
2. Contract Public Key - The public key of the issuer of the contract – NOT REQUIRED
3. Merge UTXOs - The UTXOs containing the STAS tokens to be merged (must be two UTXOs from the same owner)
4. Destination Addresses - An address that the tokens will be sent to
5. Payment UTXO - The funds in UTXO format for the transaction fee
6. Payment Private Key - The private key for the payment of transaction fees

```
1  const mergeHex = merge(  
2    bobPrivateKey, // the owner of the tokens to be merged  
3    [{  
4      tx: splitTxObj, // the first stas UTXO to be merged  
5      vout: 0  
6    },  
7    {  
8      tx: splitTxObj, // the second stas UTXO to be merged  
9      vout: 1  
10   }],  
11   aliceAddr, //destination address  
12   {  
13     txid: splitTxid, // funding utxo  
14     vout: 2,  
15     scriptPubKey: splitTx.vout[2].scriptPubKey.hex,  
16     amount: splitTx.vout[2].value  
17   },  
18   fundingPrivateKey  
19 )  
20 splitTxid = await broadcast(splitHex)
```

4.4.5.1 Mergesplit

Mergesplit can be used to take two existing STAS UTXO's (outputs from other transactions containing STAS tokens) from the same owner, combine them before splitting them again and sending them to two addresses. The split amounts can be sent to any addresses and can be split in any way as long as they match the input amount (the total from the two input UTXOs, ie the merged amount).

Mergesplit function arguments;

1. STAS Token Owner Private Key - The private key of the owner of the STAS tokens to be merged
2. Contract Public Key - The public key of the issuer of the contract --- NOT REQUIRED
3. Merge UTXOs - The UTXOs containing the STAS tokens to be merged (must be two UTXOs from the same owner)
4. Destination Address 1 - An address that the tokens will be sent to
5. Satoshi Amount 1 - The amount of satoshis to be sent to destination address 1
6. Destination Address 2 - An address that the tokens will be sent to
7. Satoshi Amount 2 - The amount of satoshis to be sent to destination address 2

8. Payment UTXO - The funds in UTXO format for the transaction fee
9. Payment Private Key - The private key for the payment of transaction fees

```

1 // some calculations to split satoshis into new amounts for split
2 const aliceAmountSatoshis = Math.floor(splitTx2.vout[0].value * SATS_PER_BITCOIN) / 2
3 const bobAmountSatoshis = Math.floor(splitTx2.vout[0].value * SATS_PER_BITCOIN) + Math.floor(splitTx2.vout[1].value * SATS_PER
4
5 const mergeSplitHex = mergeSplit(
6   alicePrivateKey, // the owner of the tokens to be merged and split
7   [{
8     tx: splitTxObj2, // the first stas UTXO to be merged
9     scriptPubKey: splitTx2.vout[0].scriptPubKey.hex,
10    vout: 0,
11    amount: splitTx2.vout[0].value
12  },
13  {
14    tx: splitTxObj2, // the second stas UTXO to be merged
15    scriptPubKey: splitTx2.vout[1].scriptPubKey.hex,
16    vout: 1,
17    amount: splitTx2.vout[1].value
18  }],
19  aliceAddr, // destination address 1
20  aliceAmountSatoshis, // amount of satoshis to be sent to destination address 1
21  bobAddr, // destination address 2
22  bobAmountSatoshis, // amount of satoshis to be sent to destination address 2
23  {
24    txid: splitTxid2, // funding utxo
25    vout: 2,
26    scriptPubKey: splitTx2.vout[2].scriptPubKey.hex,
27    amount: splitTx2.vout[2].value
28  },
29  fundingPrivateKey
30 )
31
32
33 mergeSplitTxid = await broadcast(mergeSplitHex)

```

4.4.6 Atomic swap

Atomic swap is not yet part of the js sdk, and will be released in the next weeks of January. it allows you to do a token to token swap or a sats to STAS token to sats swap.

4.4.7 Redemption

The redeem function takes all of the tokens from an existing STAS UTXO (an output from other transactions containing STAS tokens) and converts them all back to BSV and then sends them to the redeem address that was specified when the token contract was created.

Redeem function arguments;

1. STAS Token Owner Private Key - The private key of the owner of the STAS tokens
2. Contract Public Key - The public key of the issuer of the contract
3. STAS UTXO - The UTXO containing the STAS tokens
4. Payment UTXO - The funds in UTXO format for the transaction fee
5. Payment Private Key - The private key for the payment of transaction fees

```
1  const redeemHex = redeem(  
2    alicePrivateKey, // the owner of the tokens to be redeemed  
3    issuerPrivateKey.publicKey,  
4    {  
5      txid: mergeSplitTxid, // the STAS UTXO to be redeemed  
6      vout: 0,  
7      scriptPubKey: mergeSplitTx.vout[0].scriptPubKey.hex,  
8      amount: mergeSplitTx.vout[0].value  
9    },  
10   {  
11     txid: mergeSplitTxid, // The funding UTXO for fees  
12     vout: 2,  
13     scriptPubKey: mergeSplitTx.vout[2].scriptPubKey.hex,  
14     amount: mergeSplitTx.vout[2].value  
15   },  
16   fundingPrivateKey  
17 )  
18 const redeemTxid = await broadcast(redeemHex)
```

4.4.7.1 Redeemsplit

Redeemsplit essentially performs two different functions. It takes a single STAS UTXO (an output from other transactions containing STAS tokens) and splits a portion of them and sends them to upto three different addresses, the rest are converted back to BSV and sent to the redeem address that was specified when the token contract was created. As you can see below we do not provide an address for redemption. The input amount minus the output to the split destinations is sent back to the redeem address.

RedeemSplit function arguments:

1. STAS Token Owner Private Key - The private key of the owner of the STAS tokens
2. Contract Public Key - The public key of the issuer of the contract
3. STAS UTXO - The UTXO containing the STAS tokens

4. Split Destination Addresses - An array of addresses and token amounts the tokens are to be transferred to
5. Payment UTXO - The funds in UTXO format for the transaction fee
6. Payment Private Key - The private key for the payment of transaction fees

```
1 // some calculations for the split
2 const rsBobAmount = issueTx.vout[0].value / 5
3 const rsAliceAmount1 = issueTx.vout[0].value / 5
4 const rSplitDestinations = [] //the array of spllits
5 rSplitDestinations[0] = { address: bobAddr, amount: rsBobAmount }
6 rSplitDestinations[1] = { address: aliceAddr, amount: rsAliceAmount1 }
7 rSplitDestinations[2] = { address: bobAddr, amount: rsBobAmount }
8
9 const redeemSplitHex = redeemSplit(
10   alicePrivateKey, // the owner of the tokens to be redeemed and split
11   issuerPrivateKey.publicKey,
12   {
13     txid: issueTxid, // the stas utxo to be split and redeemed
14     vout: 0,
15     scriptPubKey: issueTx.vout[0].scriptPubKey.hex,
16     amount: issueTx.vout[0].value
17   },
18   rSplitDestinations, // the split destination addresses - the remaining sats are converted back to
19   { // bsv and sent to redemption address from the contract.
20     txid: issueTxid, // The funding UTXO for fees
21     vout: issueOutFundingVout,
22     scriptPubKey: issueTx.vout[issueOutFundingVout].scriptPubKey.hex,
23     amount: issueTx.vout[issueOutFundingVout].value
24   },
25   fundingPrivateKey
26 )
27 redeemTxid = await broadcast(redeemSplitHex)
```

5 How to use the API

As explained the SDK builds the transactions that are then signed and submitted to the node for validation purposes before the miners confirm the transaction from being valid and mined in the next block.

To read tokens over Whatsonchain the following STAS endpoints should be integrated.

5.1 What API functionalities do we have

5.1.1 Get all tokens

This endpoint retrieves a list of supported tokens with metadata.

HTTP Request

GET <https://api.whatsonchain.com/v1/bsv/<network>/tokens>

URL Parameters

Parameter	Description
network	The selected network, that is main or test.

5.1.2 Get token by Id

This endpoint retrieves the token metadata, which includes the contract and issuance details.

HTTP Request

GET <https://api.whatsonchain.com/v1/bsv/<network>/token/<tokenId>/<symbol>>

URL Parameters

Parameter	Description
network	The selected network, that is main or test.

tokenId	The redeem address of the token.
symbol	The symbol of the token.

5.1.3 Get address token balance

This endpoint retrieves the address of the token balance.

HTTP Request

GET <https://api.whatsonchain.com/v1/bsv/<network>/script/<address>/tokens>

URL Parameters

Parameter	Description
network	The selected network, that is main or test.
address	The balance of tokens for a given address.

5.1.4 Get script transactions

This endpoint retrieves the script of the token balance.

HTTP Request

GET <https://api.whatsonchain.com/v1/bsv/<network>/script/<scriptHash>/tokens>

URL Parameters

Parameter	Description
network	The selected network, that is main or test.
script	The balance of tokens for a given scripthash.

5.1.5 Get token transactions

This endpoint retrieves the token transactions.

HTTP Request

GET <https://api.whatsonchain.com/v1/bsv/<network>/token/<tokenId>/tx>

URL Parameters

Parameter	Description
network	The selected network, that is main or test.
tokenId	The unique ID of the token.

5.1.6 Get token output details

This endpoint retrieves the details of the valid token output or Not Found (404) for the invalid token output.

HTTP Request

GET <https://api.whatsonchain.com/v1/bsv/<network>/token/tx/<hash>/out/<index>>

URL Parameters

Parameter	Description
network	The selected network, that is main or test.
hash	The hash or transaction ID of the transaction.
index	The Output or Vout index of the token.

6 Performing Testing

There is a file called `lifecycle.test.js` that exercises a full lifecycle of a STAS token.

- There are various mocha tests located in test folder.
- Files ending `_tests` contain multiple general tests per function (eg `contract_tests.js` contains contract tests) Files with specific names contain tests targeting a specific test case (eg `mergeInvalidStasToken.js`)

All transactions are submitted to Taalnet, a private BSV blockchain that is maintained for testing STAS tokens. All tokens created can be viewed at <https://taalnet.whatsonchain.com/tokens>.

Please note that the testing library will be updated as part of the readme documentation and the defined framework and structure will likely change, as we continue the development and enhancements whilst making the transition from the public beta to a fully available public STAS SDK library in 2022.

7 Appendix

7.1 Appendix A: Standard Metadata types

Name	Type	Values
protocolid	TBD	TBD
name	string	The limit is upto 32 characters and UTF-8 standard.
tokenId	string	pubkey or address + ticker
symbol	string	The limit is from 0-9 decimal numbers.
description	string	The description of the token. The limit is upto 512 characters.
image	string	The limit is upto 512 characters.
totalSupply	integer	Unlimited but limited in terms of Sats compared to the total what can be done.
decimals	integer	The limit is from 0-8 decimal numbers.
satsPerToken	integer	Unlimited, but limited in terms of Sats compared to the total what can be done.
properties	{ "legal": { terms: '© 2020 TAAL TECHNOLOGIES SEZC\nALL RIGHTS RESERVED. ANY USE OF THIS SOFTWARE IS SUBJECT TO TERMS AND CONDITIONS OF LICENSE. USE OF THIS SOFTWARE WITHOUT LICENSE CONSTITUTES	The maximum size is 20 kb. Legal: Should be locked from not being changeable.

	<p>INFRINGEMENT OF INTELLECTUAL PROPERTY. FOR LICENSE DETAILS OF THE SOFTWARE, PLEASE REFER TO: www.taal.com/stas-token- license-agreement',</p> <p>licenseId: ...</p> <p>},</p> <p>"issuer": {</p> <p>organization: 'Taal Technologies SEZC',</p> <p>legalForm: '...',</p> <p>ISO-3166: CA,</p> <p>mailingAddress: '...',</p> <p>jurisdiction: '...',</p> <p>email: 'info@taal.com',</p> <p>},</p> <p>"meta": {</p> <p>"schemaId": "RFC 3986",</p> <p>"website": "vaionex.com",</p> <p>"legal": {</p> <p>terms: "IP terms of the nft"</p> <p>},</p> <p>"media": {</p> <p>"type": "mp4",</p> <p>"URI": "... .." } },</p>	<p>licenseId: Determined by Taal, to identify a client over an ID provided upon signing the STAS user agreement.</p> <p>Issuer information needs to comply with ICANN standard (for domain registrations).</p> <p>Meta: Dynamic rich data field.</p> <p>For example:</p> <p>With schema Id</p> <p>RFC 3986 NFT format (protocol wise a json free to be picked)</p> <p>and respective data fields.</p>
--	---	---