# Tokenized Agent Setup Guide

Brendan Lee

Tokenized is building something on a seriously different level to what I have seen from any other entrant to the token game. James Belding has a grand vision and has assembled an A grade team capable of creating something of colossal potential which behaves in a manner unlike anything a business has seen before, but which can still be described using familiar words and concepts. To understand it we must first look at the ledger that Tokenized uses as its foundation.

This is the first of a series of 5 blog posts that will explore the Tokenized smart contract architecture and take users through how to set up the necessary infrastructure to set up and run a smart contract and create a tokenized asset.

## The ledger

The Bitcoin ledger is a globally distributed data structure managed by Bitcoin miners. The Bitcoin Protocol determines a few underlying rules stipulating how users interact with the ledger, but for the intents and purposes of general use, it is reasonable to consider the ledger as a completely open place where anybody can record information that is useful to them.

An integral part of the ledger is the Bitcoin currency which can be exchanged on the ledger in Bitcoin transactions. These transactions use cryptographic hash functions and a highly secure Elliptic Curve Digital Signature Algorithm to protect the ownership rights associated with that currency.
In addition to the exchange of Bitcoins, the ledger can also be used to create tokenized representations of other asset types and allow those to be traded with the same levels of security as Bitcoins.

The fundamental difference between Bitcoins and other token types is that the bitcoin tokens are issued by Bitcoin miners to each other as rewards for working on building the ledger. They control the processes and protocols by which those tokens can be exchanged on the ledger and have the right as a collective to approve or not to approve your transaction. When something from the real world is tokenized, the tokens are issued by a controlling party. This might be a company who issues shares, a bank who tokenizes an asset or a venue who sells a ticket to a concert or an event. These tokens operate under a completely different set of rules to those which govern the exchange of Bitcoins on the Bitcoin SV network.

This is where second layer protocols are needed. A second layer protocol resides inside Bitcoin transactions, using the immutability and security of Bitcoin transactions as a means to manage the changing of ownership rights of real-world items. There are several token protocols that exist for Bitcoin however none are as complete or as ready for managing the requirements of businesses operating in lawful, regulated environments as the Tokenized Protocol. The Tokenized Protocol is also inherently more efficient at scale than competing protocols, and is also a fundamentally fairer protocol that can stop all technical forms of bad behaviour that an issuing entity could be tempted to engage in, like market manipulation or ignoring some transactions while permitting others. The request-response mechanism of the Tokenized Protocol is not only fairer for token holders, it also represents a revolution in the way that regulators (and other interested parties) can monitor for compliance with rules and regulations.

# The Tokenized Protocol

The introduction of the Tokenized Protocol means that users can now create notes on pages owned by smart contracts in order to manage the ownership rights of tokens. The smart contract uses these ledger records to manage its tokens. The issuer owns the smart contract, and dependent on the rules of the contract, has the power to send instructions to the agent that override requests from users. To police the use of these features, the total history of every smart contract can be audited by third parties.

Tokenized have a sophisticated encryption scheme that allows the contents of a contract to be kept private despite being stored in public. Every request to a smart contract is written to a contract's address for evaluation and the responses from contracts come from those same addresses. These contract addresses will serve as anchor points for business conducted by the corporations of the future.

## The Tokenized Smart Contract Agent

The tokenized Smart Contract Agent is an autonomous software application that runs on a server connected to the internet. The contract agent can be operated by the entity issuing a smart contract, or by a third-party provider as a service. The only means to interact with an agent is by writing instructions to the Bitcoin ledger. They do not take instructions via Internet Protocol giving them a very small attack surface.

Currently the smart contract agents require users to run a BitcoinSV node which communicates with the tokenized Spy Node system. This is a lightweight network monitor that the agent uses to filter network traffic to see transactions being sent to it by users. Longer term, Tokenized plans to introduce an API service for spy nodes which will remove the need for users to run their own nodes, significantly reducing the operational overhead in a scaled Bitcoin scenario.

## Entities

Entities are at the core of the Tokenized system. Entities are defined during the establishment of a smart contract and are used to build out on-chain representations of real-world legal entities and ownership structures, like public and private companies, partnerships, trusts, and even government agencies, enabling organizations to define and permissioned control to people of importance to their operation such as directors, lawyers, accountants and more.

## Smart Contracts

Tokenized smart contracts are agreements established between a real-world entity (e.g. a company or individual) and an agent which outline a set of data points and rules which the agent will use to govern interactions with the users of that smart contract. This includes datapoints describing the entity setting up the contract and outlines rules such as limits on token issuance, voting rights, and the means used to manage ownership of the contract and the assets it manages. This information is sent to the agent inside on-chain Tokenized request actions, which the agent evaluates and acknowledges in its own on-chain responses.

Smart contracts can be used to manage one or many asset types for their owners and can be established in parent-child relationships allowing for complex contract trees to be established. This is useful for corporations who might have a single umbrella contract to manage the company's shares, board of directors and more, with sub-contracts existing for diverse purposes such as delivering a user interface needing tokens, managing logistics or warranty and more.

## Assets

Assets controlled by these agents can be anything from movie tickets to shares, tokenized real estate or even national currency issued by central banks. Each asset is created with a set number of tokens to represent it. Assets can be fungible (many identical tokens) or non-fungible (single unique token) and represent almost anything that we use to exchange goods and services in the real world. Everyone with access to Bitcoin will now be able to integrate the rapid, immutable and low-cost exchange of all kinds of tokenized goods and services.

Ownership rights can be conferred upon goods, services, and equity and then managed by the token issuer in a simple and transparent way. Each contract is programmed publicly through on-chain actions detailing its purpose, who owns it, how its voting rights are managed and more. All the details needed to build a legal foundation for the establishment of an entity that holds and manages tokenized ownership rights can be established in a Tokenized smart contract.

## Using Tokenized – A beginner's guide

Today we have released a set of guides that go through the basics of setting up a Tokenized smart contract agent on your local machine, building a smart contract, and defining, issuing and distributing a set of assets. They will be officially published on a weekly basis as our Faia blog, however so people can start using the information straight away, all four posts are currently available via the following links:

# Setting up the Tokenized Smart Contract Agent

To run the Tokenized smart contract, you will need to prepare a suitable environment.

During this process, we tried to demonstrate the smart contract operating on the 3 major operating systems including:

- Windows 10
- Linux
- MacOS

## Windows 10

Since the inception of the BitcoinSV node client, the team behind the effort have focussed exclusively on developing for Linux as this is what was requested by their financial sponsors. Windows 10 has what is called 'Windows Subsystem for Linux' or WSL which allows users to install Linux variants as a subsystem to Windows.

There are caveats to this and the drive mounting systems that are used caused us several issues when trying to operate a Bitcoin node. We found that the WSL environment was unable to synchronise the BSV blockchain.

In addition, Tokenized do not have binaries for the smart contract agent or CLI utility for Windows.

For these reasons, we do not recommend setting up either BitcoinSV or Tokenized using the Windows platform at this time.

## Setting up BitcoinSV

A smart contract agent requires access to a BitcoinSV node to send and receive Bitcoin transactions on the network. Currently Tokenized do not offer a public facing node for this service so users need to have access to a node that is running with indexing and a full copy of the BSV blockchain.

## MacOS

The BitcoinSV node client is not available for MacOS at this time. At this time, we would recommend using Docker to install and run the BitcoinSV node software for MacOS. Full instructions are available thanks to the BitcoinSV node team in the following Github repository:

https://github.com/bitcoin-sv/docker-sv/tree/master

Once installed, use the following settings for your bitcoin.conf file:

```
txindex=1
server=1
listen=1
rpcuser=brendan
rpcpassword=securepassword
rpcallowip=127.0.0.1
```

`txindex=1` tells the node to index all transactions in the blockchain. This is needed to allow the node to request it's own transactions via RPC.

`server=1` tells the node to accept RPC commands

`listen=1` puts the node into listening mode, enabling any external IP to make RPC requests

The RPC settings will be used by the Tokenized agent's node monitor to retrieve its transactions from the network

## Linux

For our system, we used a Digital Ocean droplet with 6 CPUs, 16GB RAM and 320GB of storage configured with the latest version of Ubuntu however anyone running Linux on their own PC with at least 8GB of RAM, 2CPUs and 200GB of free hard drive space should also be able to do the same.

Setting up the droplet was extremely easy once our account was established. Check out:

www.digitalocean.com.

We used PuTTY to connect which is available here:

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

When your Droplet is established, and each time you log in, you will be given root access. To use the smart contract you must have a separate user setup. Create a new user as follows:

```
$ adduser username
```

You will be asked to enter a password for your new user and a few other details which you can choose to leave blank. Each time you generate a new instance of your Command Line interface, you will need to change to your new user profile which is as simple as:

```
$ su username
```

The installation of BitcoinSV node is a snap.

1. Download the most recent stable release using wget as follows:

```
$ wget https://download.bitcoinsv.io/bitcoinsv/0.2.1/bitcoin-sv-0.2.1-x86_64-linux-gnu.tar.gz
```

2. Unzip the binary

```
$ tar -xf bitcoin-sv-0.2.1-x86_64-linux-gnu.tar.gz
```

3. Create and edit your bitcoin.conf file

```
touch ~/.bitcoin/bitcoin.conf
nano ~/.bitcoin/bitcoin.conf
```

Here are my settings:

```
txindex=1
server=1
listen=1
rpcuser=brendan
rpcpassword=securepassword
rpcallowip=127.0.0.1
```

`txindex=1` tells the node to index all transactions in the blockchain. This is needed to allow the node to request it's own transactions via RPC.

`server=1` tells the node to accept RPC commands

`listen=1` puts the node into listening mode, enabling any external IP to make RPC requests

The RPC settings will be used by the Tokenized agent's node monitor to retrieve its transactions from the network

4. Start Bitcoin

```
cd bitcoin-sv-0.2.1/bin
```

```
./bitcoind
```

Once you have started Bitcoin, it will commence the process of synchronising. This process involves downloading the full blockchain, indexing all the transactions as it goes. On my home internet connection using my laptop this process took roughly 4 days.

You can check progress by typing

```
sudo ./bitcoin-cli getblockchaininfo
```

in the `~/bitcoin-sv-0.2.1/bin` folder.

Check what the latest blockheight is on www.whatsonchain.com to see how far along you are.

## Download and install Golang

Before you can use the Tokenized Smart Contract Agent you will need to install Golang. Golang is available for all major operating systems.

### MacOS

In MacOS, the following process can be used to install Golang:

#### *Download Golang*

Visit www.golang.org and download the latest MAC PKG file. Open this and follow the prompts to install the Go tools. This package installs your GO distribution to /usr/local/go and set your PATH environment variable. You'll need to restart any terminal windows you have open for the settings to take effect.

### Linux

In our Linux example, I used the following process:

#### *Install linux upgrades*

These commands will apply the latest updates to the Linux subsystem:

```
$ sudo apt-get update
$ sudo apt-get -y upgrade
```

#### *Download Golang*

This command will download version 1.13.5 of Golang, but you should check the www.golang.org website to make sure you are getting the most up-to-date version:

```
$ wget https://dl.google.com/go/go1.13.5.linux-amd64.tar.gz
```

Usually they are consistent in their naming so it suffices to simply update the version number in the file name.

#### *Extract and install*

Now that you have downloaded Golang, use the following command to extract the archive:

```
$ sudo tar -xvf go1.13.5.linux-amd64.tar.gz
```

And now move the files into the usr/local folder:

```
$ mv go /usr/local
```

### *Setup the Go environment*

Now that Go is installed, you will need to make some modifications to your Linux environment by setting the GOROOT, GOPATH and PATH variables in your ~/.profile file.

To edit, I used nano:

```
$ nano ~/.profile
```

Add these three lines to the end of the file (feel free to change the GOPATH to the location you wish to use for your projects):

```
GOROOT=/usr/local/go
GOPATH=$HOME/go
PATH=$GOPATH/bin:$GOROOT/bin:$PATH
```

Now that you have edited your `.profile` file, each time you open a new command line you can load the updated settings by typing:

```
$ . ~/.profile
```

## Ready to go

And with that, your environment should be ready to go. Next week we will look at how to install and configure the Tokenized Smart Contract Agent.

## Setting up the Tokenized Smart Contract Agent

As we discussed in our first post, the Tokenized Smart Contract Agent is an autonomous process that responds to commands written to its address on the ledger. Every request that pays the correct fees gets a response, and any request that is a correctly formatted tokenized request which asks the contract to do something which is within its defined ruleset is followed by a response action from the agent. If someone tries to do something outside the rules, the agent responds with a rejection action. All actions are encapsulated within Bitcoin transactions, giving attackers almost no surface from which to attack a tokenized smart contract.

The agent is currently still in beta so there isn't yet a simple binary to download so you will need to download and install it manually. Once again, this installation is being done in the Microsoft Windows Linux Subsystem, which has proven quite smooth and robust to use!

These instructions are applicable to users of both Linux and Windows operating systems.

### Downloading the smart contract code

Once you have installed go, downloading the code and binaries is as simple as typing:

```
go get github.com/tokenized/smart-contract/cmd/...
```

from a bash prompt. The first time this can take a while (almost half an hour for me) but once you've downloaded the full dependency set it tends to go a lot faster.

If you are keen, you can download the sourcecode and build the binaries yourself, and there are detailed instructions on how to do this at the Tokenized Github repository:

https://github.com/tokenized/smart-contract

### Smart Contract configuration

Once you have the smart contract installed you must setup the agent's configuration file. I am going to set this smart contract up to manage shares in my Coinstorage business which sells the SAFEWORDS key backup kit.

I started by copying the example config file provided with the source into the ~/.contract folder using the following command:

$ cp conf/dev.env.example ~/.contract/coinstorage.env

I then opened the file in 'nano' to make the required changes:

$ nano ~/.contract/coinstorage.env

This is what my .env file looks like now:

```
# Name of smartcontract instance.
export OPERATOR_NAME=CoinstorageGuru
export VERSION=0.1.0

# Private key in WIF format (this is not the real key 😉)
export PRIV_KEY=5K1mPd9G7oRALRietnyY2HrRFEFBDHeyHeyHeeeeeyy

# The receiving address for contract fees (Not the smart contract address)
export FEE_ADDRESS=1EeG8jTbBzKWqRNbtWjcav5VKMnJ1ZK7wf

# The cost of a request
```

```
export FEE_VALUE=1000

#The chain being used for this contract
export BITCOIN_CHAIN=mainnet

# My BitcoinSV node details
export RPC_HOST=127.0.0.1:8332
export RPC_USERNAME=brendan
export RPC_PASSWORD=securepassword

# Tell spynode which node to connect to
export NODE_ADDRESS=127.0.0.1:8333
export NODE_USER_AGENT="/Tokenized:0.1.0/"

# Checkpoint at Block 610,000
export
START_HASH="00000000000000000006728585ad384293914ee5cce36223594b07210aaef0617
2"
export NODE_STORAGE_ROOT=./tmp/contract
export NODE_STORAGE_BUCKET=standalone

# Where to store contract state
export CONTRACT_STORAGE_ROOT=./tmp/contract
export CONTRACT_STORAGE_BUCKET=standalone

export LOG_FILE_PATH=./tmp/contract/main.log
```

### Starting the agent

Once you have created the export file with the correct options for your configuration, starting the smart contract agent is as simple as executing the following command from your user directory:

```
$ source .contract/coinstorage.env && smartcontractd
```

Congratulations, you are now running the Tokenized Smart Contract Agent.

## Setting up the client tool

The tokenized package comes with a simple client tool that allows you to build and send Tokenized packets from within your CLI.

To configure this, start by creating a folder for your config file:

```
$ mkdir ~/.client
```

Now copy the example file across into your new directory:

```
$ cp ~/go/src/github.com/tokenized/src/smart-
contract/conf/cli.dev.env.example ~/.client/cli.dev.env
```

Now use nano to edit the file:

```
$ nano ~/.client/cli.dev.env
```

I used the following settings:

```
#
# Config for a standalone contract instance.
#
```

```
# the local node to connect to.
export CLIENT_NODE_ADDRESS=127.0.0.1:8333
export CLIENT_NODE_USER_AGENT="/TokenizedClient:0.1.0/"

# Sync contract from Block 610,000
export
CLIENT_START_HASH="00000000000000000006728585ad384293914ee5cce36223594b07210a
aef06172"

# Your key in WIF format (not my key)
export CLIENT_WALLET_KEY=5JfFnjW7kAFVZCckJMLgKEA6ATJgdnY9HeyHeyHeeeeyyyyyyy

# Address of contract in standard bitcoin format
export CLIENT_CONTRACT_ADDRESS= 1N9nJMPfJGnhcWrYAtY2DvsHb3cXfVmsTq

# Where to store state files
export CLIENT_PATH=./tmp/client

export CLIENT_LOG_FILE_PATH=./tmp/client/main.log

export BITCOIN_CHAIN=mainnet
```

NOTE: When inputting the CLIENT_CONTRACT_ADDRESS make sure to use the COMPRESSED ADDRESS format as the Smart Contract will not see any transactions sent to the address that corresponds to the uncompressed address.

With this file, we can now execute commands via the Smartcontract tool to build and send on-chain actions to interact with the agent.

You can set the CLIENT_START_HASH to a value from a recent block in order to reduce the time it will take to synchronise. For this project, we will start from block 610,000 which was mined on the 23rd of November 2019.

## Funding and synchronising the Client

To prepare for the creation of your smart contract, your client must have funds. Simply send a small amount (say 10c) to the address that corresponds to the WIF in your client definition file. Once you have sent the funds, execute the following from the command line in your user directory:

```
$  source ~/.client/cli.dev.env && smartcontract sync
```

It took about 5 minutes to synchronise with the funds I sent.

Next week we will look at how to fund our administrative address and communicate with the smart contract to create a Smart Contract.

## Creating a Tokenized Smart Contract

Depending on the type of token being issued, contracts require varying levels of documentation and complexity. The smart contract is adaptable and can represent anything from very simple agreements between friends right up to the definition of corporate or state level bodies.

The smart contract is defined in a request action called a Contract Offer. To learn more about how a Contract Offer is structured, check out the documentation at:

https://tokenized.com/docs/protocol/actions#action-contract-offer

The contract offer action contains all necessary information to establish a legal contract including the legal text of the contract, information on the type of contract, links to supporting documentation and more.

Importantly the contract includes data on the issuer that is asking the agent to establish the content. The issuer data is contained in a compound data structure called an 'Entity' which can be expanded to define a company structure including multiple personnel, addresses and other pertinent details as required to establish the legal identity for the contract.

For more information on the Entity type, have a look at the docs:
https://tokenized.com/docs/protocol/actions#type-entity

The contract offer can also include details such as data on the contract operator (also an entity which can be a third party distinct from the contract issuer), details of any oracles that might be required to confirm the details of the administrator, a blockheight for the oracle to sign, fee settings for the contract and any rules needed for asset management such as the quantity of different asset types that can be managed, voting rights and more.

### Building the Contract Offer action

Before we can start using the smart contract, we must set up the contract offer that we are going to send to the agent. This is prepared as a `.json` file. For the contract offer I will be making, my file looks like the following:

```
{
      "ContractName" : "Coinstorage Pty. Ltd.",
      "BodyOfAgreementType" : 2,
      "BodyOfAgreement" :
            "5468697320636f6e7472616374206d616e616765732074686520f776e6572
            736869702072696768747320746f20436f696e73746f7261676520507479204
            c746420616e6420616c6c206f7468657220746f6b656e732073656420666f
            7220707572706f736573207375636820617320c6f79616c74792070726f677
            2616d732c207072697a657320616e6420636f75706f6e206469737472696275
            74696f6e2e",
      "ContractType" : "Shareholder Agreement and Token Management",
      "Issuer" : { "Name" : "Coinstorage Pty Ltd", "Type" : "C", "Street" :
            "P.O. Box 89", "City" : "Toowong", "TerritoryStateProvinceCode"
            : "QLD  ", "PostalZIPCode" : "4066        ", "EmailAddress" :
            "info@coinstorage.guru",  "Management" : [ {"Type" : 5, "Name"
            : "Brendan Lee"} ], "DomainName" : "www.coinstorage.guru" },
            "IssuerLogoURL" : "https://coinstorage.guru/wp-
            content/uploads/2018/02/coinstorage-logo-verson-3.png",
      "ContractFee" : 1000,
      "MasterAddress" : "20c8abe70379b2b6801bf38d477d9249dfc2a9ba9c"
}
```

This is a very simple contract set up in the name of my company, Coinstorage Pty Ltd.

The BodyOfAgreement type indicates that the wording of the contract is in the Tokenized Body of Agreement Format which for the purposes of this contract is just the following text encoded in hex:

```
"This contract manages the ownership rights to Coinstorage Pty Ltd and all other tokens used for purposes such as loyalty programs, prizes and coupon distribution."
```

While this is a simple example, it is possible to create full legal documentation for a smart contract in Markdown format and convert that text to hex for insertion in your smart contract.

The contract type is specified by the issuer and should be an appropriate description of what the contract does.

Issuer refers to the real-world owner of the smart contract and the responsible party for any tokens it issues. In this example, it has the legal details of my company, Coinstorage Pty Ltd and lists myself as CEO.

ContractFee sets the price for request actions to be looked at by the contract.

MasterAddress gives an address from which a command to migrate the contract to a new address can be issued. This is a useful backup in cases where a contract's private key has been lost or compromised. This is the public key in HEX format. To generate this value, use the following command in your user directory:

```
$ smartcontract convert yourKeyHere
```

and adding `20` to the start to tell the tool that it is a P2PKH address.

Keep in mind that fixed width fields such as TerritoryStateProvinceCode must be the expected length which is achieved here by adding spaces.

While this contract is quite simple, the structure of the Contract Offer and entities allows for corporation or state level structures to be defined.

## Checking the Contract Offer

Before sending the Contract Offer, check that your JSON builds properly using:

```
$  smartcontract build c1 ~/coinstorage/C1_coinstorage.json  --hex
```

If you JSON file has no issues, it should return the hex corresponding to your contract offer.

Now build the TX using:

```
$  smartcontract build c1 ~/coinstorage/C1_coinstorage.json  --tx
```

This should return the full tx structure including the funding coin and change output.

## Send your contract offer

Send your contract offer to the network using:

```
$  smartcontract build c1 ~/coinstorage/C1_coinstorage.json  --tx --send
```

For my contract, the corresponding TXID is:

b6627a3c734db0e7672c19bf9308cb7550a617c2288fde386125949573e87322

I was very surprised how fast it was. As soon as I could refresh the page on www.whatsonchain.com I saw the smart contract's response which has TXID:

c7c5d1bf05c291fdc6b8c9089444b45cf8841613eec602c207516c5d48619a2c

You can see that it has properly mirrored all the settings from my transaction.

To update your tool to see the response transation use the sync command:

```
$  source ~/.client/cli.dev.env && smartcontract sync
```

## Congratulations

Congratulations, you have created a Smart Contract using the Tokenized Smart Contract agent. Next week we will look at how to create and send assets using the platform.

## Tokenizing Assets

Tokenized smart contracts are for the purpose of managing real world assets in a way that reflects the real-world way in which that asset is being managed.

The first version of the Tokenized Smart Contract Agent supports the following asset types with the associated Asset Code in brackets:

- Membership Tokens (MEM)
- Common Shares (SHC)
- Currency (CUR)
- Coupons (COU)
- Loyalty Points (LOY)
- Admission Tickets (TIC)
- Casino Chips (CHP)

While these 7 types are supported by the beta version of the agent, the Tokenized Protocol can support any need and the team have a list of over 50 different types of token being prepared including coverage of all general financial products used in retail and enterprise banking, investment and more.

For more information on the asset types supported by the current Tokenized agent, please visit:

https://tokenized.com/docs/protocol/assets

For this example, I will create a set of 5 common shares to manage ownership rights for Coinstorage Pty. Ltd. Keep in mind, these are being created using the tokenized.test protocol identifier and as such should not be considered binding.

### Asset Structure

Assets are defined using a structure held in the Asset Payload. The payload contains all the necessary information that an asset of this type would be likely to need. This can include information on what the tokens are, how they are to be used and more.

### Creating the Asset Payload

Every different asset type has a payload which specifies fields that are specific to that type of asset. These can include any type of information relevant to the management of that asset.

To learn more about the payloads for established asset types, see:

https://tokenized.com/docs/protocol/assets#field-aliases

The Smart Contract tool can build the hex payload we need to create the asset. First we need a JSON file containing the necessary information. I created the following:

```
$ nano ~/coinstorage/csgru.json
```

The definition needs only the ticker and description, with the date and ISIN left blank:

```
{
      "Ticker" : "CSGRU",
      "Description" : "Founders shares in Coinstorage Guru"
}
```

Now use the Smart Contract tool to build the hexadecimal payload:

```
$ smartcontract build SHC coinstorage/csgru.json --hex
Asset : SHC
12054353475255222346f756e6465727320736861726573206e20436f696e73746f726167652047757275
```

This payload can now be inserted into the Asset Definition action.

## Building the Asset Definition Action

The Asset Definition action is assembled in a JSON file. I created the following:

```
$ nano ~/coinstorage/A1_coinstorage.json
```

Assets are created by sending an Asset Definition action (A1) to the Smart Contract agent using an established smart contract. Learn more about this action here:

https://tokenized.com/docs/protocol/actions#action-asset-definition

Similarly, to the Contract Offer, the data must be contained in a .JSON file ready to be written into a tokenized transaction.

This is the layout of my JSON file:

```
{
      "TransfersPermitted" : true,
      "EnforcementOrdersPermitted" : true,
      "VotingRights" : true,
      "TokenQty" : 5,
      "AssetType" : "SHC",
      "AssetPayload" :
            "12054353475255222846f756e6465727320736861726573206e20436f69
            6e73746f72616765205074792e204c74642e"
}
```

Transfer Permitted allows this token to be transferred by third parties. This is good for transferrable assets but should be set to FALSE for things such as memberships. Where transfers are not permitted, the token holders will always be able to send the token back to the contract's administrative address.

EnforcementOrdersPermitted allows the issuer to execute enforcement orders such as freezing and confiscating the tokens.

VotingRights means that a holder of one of these tokens has a right to vote in company decisions.

TokenQty is set to 5 to create our 5 assets, with AssetType set to SHC for common share.

## Creating the transaction

To define the Tokenized action, we must have access to the Tokenized Smart Contract utility via Putty.

Check the status of your Bitcoin node, and sync the Smart Contract tool.

```
$ smartcontract sync
```

Now, build the tx:

```
$ smartcontract build A1 coinstorage/a1_coinstorage.json --tx
```

The process asks how many voting systems there are in the contract, and for this there is just 1.

The TX builds correctly, so the hex is tried next:

```
$ smartcontract build A1 coinstorage/a1_coinstorage.json --hex
```

As this looks good, our third command builds and sends the tx to the network:

```
$ smartcontract build A1 coinstorage/a1_coinstorage.json --tx --send
```

Checking https://whatsonchain.com/address/1N9nJMPfJGnhcWrYAtY2DvsHb3cXfVmsTq we can see almost immediately that the Smart Contract Agent has responded with an A2 – Asset Creation action which means it has acknowledged our request as a correctly formatted request within the established ruleset of the Smart Contract and has created the assets.

Click on 'Hex' in the decode window and copy the text. Now use the parse command to see what the contract created:

```
$ smartcontract parse
006a02bd000e746573742e746f6b656e697a6564041a0241324c6d0a20bbf16c62ed6505533
30329f2db623bdb740f58d1bae21e6af3bf0e314e523d062001300138016005 6a0353484372
311205435347525522284 66f756e6465727320736861726573 20696e20436f696e73746f726
16765205074792e204c74642e800193a6989e839181f115

Uses Test Protocol Signature
Action type : A2

```
{
    "AssetCode":
      "bbf16c62ed650553330329f2db623bdb740f58d1bae21e6af3bf0e314e523d06",
    "TransfersPermitted": true,
    "EnforcementOrdersPermitted": true,
    "VotingRights": true,
    "TokenQty": 5,
    "AssetType": "SHC",
    "AssetPayload":
      "120543534752552228466f756e6465727320736861726573 20696e20436f696e7374
      6f72616765205074792e204c74642e",
    "Timestamp": 1576827802563777299
}
```

```
{
    "Ticker": "CSGRU",
    "Description": "Founders shares in Coinstorage Pty. Ltd."
}
```

The assets are handed to the issuer, which in this case is Coinstorage Pty Ltd.

In our next and final piece, we will look at how to transfer these assets between parties.
```

## Transferring assets between parties

All Tokenized transfers are handled using the T1 – Transfer action which can manage transfers of all types with multiple parties and assets.

https://tokenized.com/docs/protocol/actions#action-transfer

I am going to transfer 1 of the 5 shares to an address I have created for the demo.

I created the following JSON file to build the transaction:

```
$ nano ~/coinstorage/T1_coinstorage.json
```

The file contains the following info:

```
{
      "Assets" : [
            {
                  "ContractIndex" : 0,
                  "AssetType" : "SHC",
                  "AssetCode" :
                        "bbf16c62ed650553330329f2db623bdb740f58d1bae21e6af3
                        bf0e314e523d06",
                  "AssetRevision" : 0,
                  "AssetSenders" : [{"Index" : 0, "Quantity" : 1}],
                  "AssetReceivers" : [{
                              "Address" :
                                    "20c7bd4b65fb23514e3d80373b9828fdf13f01
                                    c90d",
                              "Quantity" : 1
                        }]
            }
      ]
}
```

This defines the transfer action by listing the asset code for our create asset. We are using asset revision 0 as we haven't modified it yet. We are sending one token to an address, or eventually any valid Bitcoin locking script, I created to receive it.

We specify that the sender is sending 1 asset and the receiver is receiving 1 asset.

The address is converted to Hexadecimal using:

```
$ smartcontract convert YourAddressHere
```

Append 20 to the front of the hexadecimal string that is generated to create the address.

Now that the transfer action is built, test it with:

```
$ smartcontract build T1 coinstorage/T1_coinstorage.json --tx
```

If the tx validates, send it using:

```
$ smartcontract build T1 coinstorage/T1_coinstorage.json --tx --send
```

The transfer action request transaction (348249e7ac0acae0b328730dafc63d6655d33f0e8b54371b29f8f47980637f87) is immediately followed by the response (dfc4d02969e2812edfb79f262383bce09a2696ba41eb90b34fa60b8250a4144d) which contains the settlement action which moves the token into my new wallet.

This token can now be validated by looking at my receiving address