Smart Contract Development with FAB

This document provides information on what you need to know to get started with development of smart contracts on *Fast Access Blockchain* (also known as FAB). It assumes some familiarity with Bitcoin and Ethereum, and is meant to express the similarities and differences between FAB and these other chains, and to translate one's understanding of these systems to the FAB ecosystem. FAB is similar to Bitcoin, but is upgraded for transaction outputs allow include instructions for Ethereum-style smart contract operations.

For writing and testing the code off-chain, an online tool such as Remix IDE can be used to quickly get started with writing smart contracts, while a variety of desktop tools can be used if the time is taken to set them up.

For deploying and calling the smart contracts, along with other essential operations, this document outlines the process of forming and signing transactions, and describes usage of the FAB API to submit transactions and query the blockchain. For those that prefer RPC over the API, RPC documentation will be added at a later time.

Smart Contracts

Writing smart contracts

Smart contracts are written in a programming language called Solidity. For compatibility reasons, we suggest using version 0.8.0 of solidity if deploying on the FAB network. Some newer OPCODEs from later versions may not be supported on FAB. For information on Solidity see https://docs.soliditylang.org/en/v0.8.0/

To build and test your contracts, you can use a web-based IDE such as Remix (https://remix.ethereum.org/), or EthFiddle (https://ethfiddle.com/). This provides tools for writing the code, compiling it, deploying it into an isolated (off-chain) test environment, and more. Picking one of these is useful if you are trying to start writing a smart contract without having to spend too much time setting up your environment for writing smart contracts.

For desktop programming environments, you can use Visual Studio Code, Atom, or JetBrains, and supplement them with IDE-specific support tools for Solidity and blockchain development in general. Other useful development tools that are worth looking into for you smart contract development are Truffle, Ganache and Tenderly, if you wish to use them.

Token Contracts (FRC20)

Token contracts often use a common interface to standardize interactions with similar tokens. This means implementing a set of required functions (in addition to whatever other functions you would like) in order to ensure a basic shared functionality. For a typical fungible token, it is recommended to create an FRC20 token, which matches the ERC20 token standard popular on Ethereum. For an example contract based on this standard, see https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

function name() public view returns (string)

function symbol() public view returns (string)

function decimals() public view returns (uint8)

function totalSupply() public view returns (uint256)

function balanceOf(address _owner) public view returns (uint256 balance)

function transfer(address _to, uint256 _value) public returns (bool success)

function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)

function approve(address _spender, uint256 _value) public returns (bool success)

function allowance(address _owner, address _spender) public view returns (uint256 remaining)

Example Workflows in FAB Smart Contract Development

This page is meant to guide the reader through the document chronologically depending on the task they wish to perform. For more detail, refer to the sections mentioned in parentheses.

I. **Preparation**
   1. Generate an ECDSA key pair following the process in BIP32. You may generate multiple public keys, but be careful when using smart contracts that check the sender. It may be prudent to select only the first one for use with contracts. (instructions not included).
   2. Determine the FAB address from the public key chosen. (see *Determining FAB Address*)
   3. Obtain some FAB by having some source send it to your FAB address.
   4. Check your balance to that you now have some FAB at your address. (see *getbalance API*)

II. **Contract Creation**
   1. Write the code for a smart contract in Solidity and compile it. (see *Writing Smart Contracts*).
   2. Compile smart contract code and obtain ABI and bytecode. (see *Deploying a Smart Contract*)
   3. Get UTXOs of the address. (see *getutxos API*)
   4. Prepare and sign a contract creation transaction. (see *Deploying a Smart Contract*)
   5. Submit the contract creation transaction to the FAB network. (see *postrawtransaction API*)
   6. Check the transaction receipt to see if it was successful and to get the contract address. (see *gettransactionreceipt API*)

III. **Interact with the contract** (state-changing)
   1. Get UTXOs of the address. (see *getutxos API*)
   2. Prepare and sign a sendtocontract transaction. (see *Sendtocontract Transactions*)
   3. Submit the sendtocontract transaction to the FAB network. (see *postrawtransaction API*)
   4. Check the transaction receipt to see if it was successful. (see *gettransactionreceipt API*)

IV. **Read data from the contract** (not state-changing, free)
   1. Perform a contract call. (see *Contract Calls* and then *callcontract API*)
   2. Decode the returned abi-encoded data. (instructions not provided. See Solidity documentation).

V. **Send FAB**
   1. Get UTXOs of the address. (see *getutxos API*)
   2. Prepare and sign a send transaction (see *Send Transactions*)
   3. Check for at least one confirmation to confirm that the transaction has been mined (see *gettransactionjson API*)

Deploying a contract

Once your Solidity source code is ready, you must compile your contract. This process should get you the ABI and the bytecode.

ABI – To be more precise, this is actually a JSON interface that *describes* the ABI (Application Binary Interface). It a large JSON object specifying the various methods and their inputs, outputs, and other information.

bytecode - A very long hex string which contains an abi-encoded version of the compiled contract code. At this point we are supplying no constructor parameters. If the constructor of the contract has no parameters, this code may be directly used to deploy the contract. If constructor parameters are required, see below on how to include them.

constructor parameters - This depends on the contract code, but if a constructor requires parameters, they must be provided when deploying. If you use a tool for this, it will likely require you to specify the parameter list (i.e. the types) and the arguments you are using, and will return an abi-encoded hex string.

Combine the bytecode with an abi-encoded set of parameters based on the ABI and the arguments to be passed in the constructor. You can find a tool or library to help with this, or look up the specification in detail at https://docs.soliditylang.org/en/v0.8.0/abi-spec.html . If you already have an abi-encoded string of the arguments, you can simply append it to the bytecode, combining them into one hex string. Once combined, you will have a long (or potentially extremely long) hex string that will be used when building one of the UTXOs for the contract deployment transaction.

The following assumes some familiarity with Bitcoin and Ethereum-style blockchains. FAB is a Bitcoin-like system that incorporates Ethereum-style smart contract elements within Bitcoin-style UTXOs. If there are aspects of these systems mentioned that you are not familiar with, consider consulting outside sources of information. Unlike Bitcoin, however, it may be convenient to use a single FAB address as part of your wallet, to make managing smart contracts easier.

Below is a description of some of the terms used in the process of creating the transaction, and some suggested values. Suggested values are meant as starting point, and may be modified to better suit your needs.

gas limit – (suggested: 4,000,000) - Maximum amount of gas to supply towards the execution of the smart contract transaction. If there is not enough to complete the transaction, it will fail. A contract deployment will typically require more than a sendtocontract transaction. Will also depend on the number of operations being done and the data to be stored.

gas price –(suggested: 40) - The price to be paid for each unit of gas.

value – (suggested: 0) – The amount of FAB to send to be sent as part of the contract-related UTXO. If the contract does not require sending FAB as part of its construction, will be 0.

bytecode – The bytecode of the contract, assumed to already be combined with any constructor arguments required as described earlier in this section.

bytes per input – (suggested: 150) – Used for approximating transaction size, for fee calculations.

satoshis per byte – (suggested: 100) – The price that is being paid for each byte of the transaction, not including the contract itself (the fees for that are counted differently).

UTXOs – The set of unspent transaction outputs belonging to the sender's address. A subset of these are to be selected, and the sum of their values should be enough to cover all FAB sent and all applicable transaction fees. Fetch a list of these via FAB API (getutxos) or RPC (listunspent) if available.

Derived and calculated values:

contract script – an encoded value generated using a btc script builder, such as script.compile from bitcoinjs-lib (assume we import bitcoinjs-lib as Btc). Some opcodes are FAB-specific.
Btc.script.compile([
    84 (representing OP_4, which indicates EVM version)
    gas limit (converted to a buffer)
    gas price (converted to a buffer)
    bytecode (without any 0x prefix , then converted to a buffer)
    193 (representing OP_CREATE)
])

transaction size = (# of inputs * bytes per input) + (# of outputs * 34) + 10

transaction fee – Calculated value of fees to go to the miner in FAB.
    Fee = (transaction size) * (satoshis per byte)
        + (gas price * gas limit + length of contract script output string * 10)/10^8

change amount – Amount to return to the sender as change.
    change = sum of all input values – FAB sent to recipients – transaction fee

ECDSA key pair - A private key and public key pair generated according to BIP 32.

Making the Transaction:

Forming and signing a FAB transaction greatly resembles Bitcoin, but with potential differences in the data supplied in the transaction outputs.

For the purposes of this guide, instructions will be given in relation to the npm library bitcoinjs-lib 6.0.1 (referred to as Btc below).

It uses a class called Psbt (Partially Signed Bitcoin Transaction, which we instantiate below as Txbuilder).

However, other libraries or custom code can be used to reproduce the same workflow as long as the underlying process is similar. The transaction format, inputs, outputs, and signing and general formulation is similar to Bitcoin. The differences lie in the data supplied in the transaction outputs (for contract-related transactions) and the method of calculating an address from a public key (see the section called *Determining FAB Address***).**

1. Initialize a transaction builder with the standard **Bitcoin** network configuration.
   ```
   Txbuilder = new Btc.Psbt(Btc.networks.bitcoin)
   ```
2. Add a number of inputs from UTXOs, with enough combined value to cover the transaction fee plus any FAB sent in any of the intended outputs. Must include at least one input. Some libraries may require a full raw transaction to be supplied for the input, from the transaction the that created the UTXO used as an input.
   ```
   Txbuilder.addInput({ "hash": utxo.txid,
   "index": utxo.index,
   "nonWitnessUtxo": utxo.rawTxofOriginatingTransaction,
    })
   ```
3. Add a change output (which returns extra FAB funds back to sender) unless the amount returned to sender would be 0 or near 0. If the value of an output is extremely small, the network may consider it as "dust" and reject the transaction.
   ```
   Txbuilder.addOutput({"address": sender address, "value":
   change amount)
   ```
4. Add an output with the contract script instead of an address.
   ```
   Txbuilder.addOutput("script": contract script,
   "value": value)
   ```
5. Sign each transaction input in a Bitcoin-like way (with ECDSA key pair)
   ```
   Txbuilder.signInput(input index, key pair )
   ```
6. Finish building the transaction.
   ```
   Txbuilder.finalizeAllInputs()
   Txbuilder.extractTransaction(true).toHex()
   ```
7. Submit to the FAB network via FAB API (postrawtransaction) or rpc (sendrawtransaction) if available.

After deployment, the transaction receipt to checked to see if it has been deployed successfully, and to verify the address of the new contract. This can be done via FAB API (gettransactionreceipt) or RPC (gettransactionreceipt) if available.

Sendtocontract Transactions

Calling a function that changes the state of a contract requires a sendtocontract transaction. Below are some values you will need to prepare for this type of transaction.

abi-encoded function call data – this is a hexadecimal string comprised of two parts. The first 8 characters are a truncated sha3 hash of the function signature for the smart contract function to be called. The remainder of the string will be an abi-encoded argument list as outlined in the Solidity language's documentation, https://docs.soliditylang.org/en/develop/abi-spec.html . You may use any Ethereum-related library such as Web3 or similar to help prepare the appropriate argument list. Note that unlike Ethereum, hexadecimal strings generally should not start with an '0x' prefix in FAB.

gas limit – (suggested: 800,000. Can be more or less depending on the complexity of the contract function) - Maximum amount of gas to supply towards the execution of the smart contract transaction. If there is not enough to complete the transaction, it will fail. A contract deployment will typically require more than a sendtocontract transaction. Will also depend on the number of operations being done and the data to be stored.

gas price –(suggested: 40) - The price to be paid for each unit of gas

value – (suggested: 0) – The amount of FAB to send to be sent to the contract as part of the function call. If the function being called is not a *payable* function, you will generally want this to be 0.

bytes per input – (suggested: 150) – Used for approximating transaction size, for fee calculations.

satoshis per byte – (suggested: 100) – The price that is being paid for each byte of the transaction, not including the contract call itself (the fees for that are counted differently).

UTXOs – The set of unspent transaction outputs belonging to the sender's address. A subset of these are to be selected, and the sum of their values should be enough to cover all FAB sent and all applicable transaction fees. Fetch a list of these via FAB API (getutxos) or RPC (listunspent) if available.

function call script – an encoded value generated using a btc script builder, such as Btc.script.compile from bitcoinjs-lib. Some opcodes are FAB-specific.

84 (representing OP_4, which indicates EVM version)
gas limit (converted to a buffer)
gas price (converted to a buffer)
abi-encoded function call data (without 0x prefix, then converted to a buffer)
contractAddress (hex version of FAB address converted to a buffer)
194 (representing OP_CALL)

transaction size = (# of inputs * bytes per input) + (# of outputs * 34) + 10
transaction fee – Calculated value of fees to go to the miner in FAB.

Fee = (transaction size) * (satoshis per byte)
+ (gas price * gas limit + length of contract script string)/10^8

change amount – Amount to return to the sender as change.
change = sum of all inputs – FAB sent to recipients – transaction fee

Making the Transaction:

The process of forming and signing a FAB transaction greatly resembles Bitcoin, but with some differences in the data that can be supplied in the transaction outputs.

For the purposes of this guide, instructions will be given in relation to the npm bitcoin library bitcoinjs-lib 6.0.1 (referred to as Btc below).

It uses a class called Psbt (Partially Signed Bitcoin Transaction, which we instantiate below as Txbuilder).

However, other libraries or custom code can be used to reproduce the same workflow as long as the underlying process is similar. The transaction format, inputs, outputs, and signing and general formulation is similar to Bitcoin. The differences lie in the data supplied in the output (for contract-related transactions) and the method of calculating an address from a public key (see the section called *Determining FAB Address*).

1. Initialize a transaction builder with the standard **Bitcoin** network configuration.
   ```
   Txbuilder = new Btc.Psbt(Btc.networks.bitcoin)
   ```
2. Add a number of inputs from UTXOs, with enough combined value to cover the transaction fee plus any FAB sent in any of the intended outputs. Must include at least one input. Some libraries may require a full raw transaction to be supplied for the input, from the transaction the that created the UTXO used as an input.
   ```
   Txbuilder.addInput({ "hash": utxo.txid,
   "index": utxo.index,
   "nonWitnessUtxo": utxo.rawTxofOriginatingTransaction,
    })
   ```
3. Add a change output (which returns extra FAB funds back to sender) unless the amount returned to sender would be 0 or near 0. If the value of an output is extremely small, the network may consider it as "dust" and reject the transaction.
   ```
   Txbuilder.addOutput({"address": sender address, "value":
   change amount)
   ```
4. Add an output with the function call script instead of an address.
   ```
   Txbuilder.addOutput("script": function call script,
   "value": value)
   ```
5. Sign each transaction input in a Bitcoin-like way (with ECDSA key pair).
   ```
   Txbuilder.signInput(input index, key pair )
   ```
6. Finish building the transaction.
   ```
   Txbuilder.finalizeAllInputs()
   Txbuilder.extractTransaction(true).toHex()
   ```
7. Submit to the FAB network via FAB API (postrawtransaction) or rpc (sendrawtransaction) if available.

After deployment, you can check the transaction receipt to check if it has been deployed successfully, and to verify the address of the new contract. You can do this via FAB API (gettransactionreceipt) or RPC (gettransactionreceipt) if available.

Contract Calls

Calling a read-only function on a contract does not require a transaction, and requires no transaction fee. Its format is similar to a transaction, and requires the following:

- The field representing the contract address is present.

- The “data” field is present, and contains an abi-encoded instruction that contains the combined information of a hash of the function signature (4bytes) followed by a set of arguments to be passed to the contract. For details, see here: https://docs.soliditylang.org/en/develop/abi-spec.html.

For ways to call the contract, use the FAB API (callcontract) or RPC (callcontract) if available.

Send Transactions

Send transactions are standard, non-contract transactions that simply involve sending an amount of FAB to a desired address. They are done much in the same way as Bitcoin transactions. To make one, follow similar steps to “deploying a contract” or “sendtocontract” transactions, but instead of doing a special contract-related output for step 4, simply use a base58 recipient FAB address instead of a btc-script-encoded contract-related message. You may also add additional outputs different of recipients in the same transaction, as long as the input amounts still cover all the outputs plus the transaction fee.

Determining FAB Address

A single FAB address can have two different forms depending on where it is used. These two forms are fundamentally the same address. One is simply the base58 representation, like a Bitcoin address though derived differently. A truncated hexadecimal version of this address is used to form something similar to an Ethereum-style address for interacting with FAB's Solidity-based smart contract system. Especially in the case of interacting with smart contracts, it can be useful to use the same address as a contract will often have no way of knowing that two different addresses come from the same private key.

To derive a base58 FAB address:

1. Begin with a compressed ECDSA public key generated as part of a key pair similarly to Bitcoin and other blockchains.
2. Apply a SHA256 hash on the compressed public key.
3. Apply a RIPEMD-160 hash on the resulting hash.
4. Ensure the output is in the base58 format, otherwise make sure to apply the appropriate conversion

To derive the hexadecimal FAB address (for contract addresses, or to be used as part of a contract parameter):

1. Begin with the base58 FAB address.
2. Convert from base58 to hexadecimal format, making sure to remove any 0x prefix.
3. Remove the first byte (2 characters) and last 4 bytes (8 characters) from the previous result. The result here should be a string of 40 hexadecimal characters.

The FAB API

The main api for FAB related requests can be found at https://fabprod.fabcoinapi.com . This URL serves as a base URL for the api, and includes some documentation for available routes if you visit the page itself.

Below is a selection of routes provided the API that may be useful when developing, deploying, and managing smart contracts.

getbalance API

Returns the FAB balance of an address, in Liu's (1 FAB = 10^8 Liu's). "balance" and "confirmedBalance" both represent the sum of unspent transaction outputs from transactions with at least 1 confirmation. "unconfirmedBalance" represents the sum of transaction outputs of *pending* transactions.

Request:

GET request - /getbalancev2/<address>
  Params:
    <address> string - a valid base58 FAB address

Response:

```
{
  "address": <address>,
  "balance": <balance>,
  "confirmedBalance": <confirmedBalance>
  "unconfirmedBalance": <unconfirmedBalance>
}
```

<address> - base58 string – the address supplied in the request.

<balance> - decimal string - sum of UTXOs from transactions with at least 1 confirmation.

<confirmedBalance> - decimal string - same as "balance".

<unconfirmedBalance> - decimal string - sum of transaction outputs of *pending* UTXOs.

getutxos API

Returns unspent transaction outputs of a given FAB address.

Request:

GET request - /getutxosv2/<address>
  Params:
    <address> string - a valid base58 FAB address

Response:

```
 [{
   "txid": <txid>,
   "idx": <idx>,
   "value": <value>
   "isConfirmed": <isConfirmed>
 },
 ...
 ]
```

<txid> hex string – Hash of the transaction containing the transaction output.

<idx> number – Index of the transaction output in the transaction containing it.

<value> number - Quantity of FAB in the transaction output.

<isConfirmed> - boolean – *true* if containing transaction is already mined, otherwise *false* (the transaction is pending).

callcontract API

Returns the result of a call to a non-state-changing function for a given smart contract.

Request:

POST request - /callcontract
  Params: N/A
  Body:
   {

     "address": <address>,
     "data": <data>,
     "sender": <sender>,
     "gas": <gas>
   }

<address> hex string – hexadecimal (without 0x prefix) 20 byte FAB address of the smart contract to be called.

<data> hex string - ABI-encoded string (without 0x prefix) containing information of the function signature identifier and supplied parameters for the contract call. See https://docs.soliditylang.org/en/develop/abi-spec.html for more info.

<sender> (optional) hex string – hexadecimal (without 0x prefix) representation of a FAB user address (i.e. not a contract address). This field may be omitted unless the function specifically requires it.

<gas> (optional) string – a functional gas limit for the call. Since a contract call does not change the state of the blockchain, and thus does not charge a fee, this field is rarely needed.

Response:

```
 {
   "address": <address>,
   "executionResult": {
         "gasUsed": <gasUsed>,
         "excepted": <excepted>,
         "newAddress": <newAddress>,
         "output": <output>,
         "codeDeposit": <codeDeposit>,
         "gasRefunded": <gasRefunded>,
         "depositSize": <depositSize>,
         "gasForDeposit": <gasForDeposit>,
       },
   "transactionReceipt": {
         "stateRoot": <stateRoot>,
         "gasUsed": <gasUsed>,
         "bloom": <bloom>,
```

```
        "log": [ ]
      },
   "comments": <comments>
  }
```

<gasUsed> - number – a measurement of the cost of execution based number and type of operations involved. For a contract call, it does not reflect an actual fee to be paid.

<excepted> - string – "None" if successful. Otherwise, the name of some exception type.

<newAddress> - hex string – the address of the contract, as shown in the original request.

<output> - hex string - ABI encoded output of the method called. The *main value of interest* for callcontract. Will need to be decoded base on knowledge of the return types of the function called.

<codeDeposit> - number – Not relevant to callcontract.

<gasRefunded> - number – Not relevant to callcontract.

<depositSize> - number – Not relevant to callcontract.

<gasForDeposit> - number – Not relevant to callcontract.

<stateRoot> - hex string - ABI encoded output of the method called.

<gasUsed> - number – a measurement of the cost of execution based number and type of operations involved. For a contract call, it does not reflect an actual fee to be paid.

<bloom> - hex string – Not relevant to callcontract.

<log> - Array of logs - Not relevant to callcontract.

<comments> - hex string - Not relevant to callcontract.

postrawtransaction API

Submits a transaction to the FAB network.

Request:

POST request - /postrawtransaction
  Params: N/A
  Body:
        {
                "rawtx": <rawTransaction>
        }

<rawTransaction> string - The transaction to be submitted. This transaction must already be signed.

Response:

```
  {
    "txid" : <transactionHash>,
    "comments" : <comments>,
  }
```

or

```
  {
    "Error" : <error>
  }
```

  <transactionHash> hex string - If successfully submitted to the kanban network, result will return a hash of the transaction submitted. This can be used to identify the transaction when looking it up.

<comments> string – Not relevant to postrawtransaction.

<error> string – A message describing an error that prevented the transaction from submitting successfully. If an error code is provided, error codes will often coincide with their Bitcoin error counterparts.

gettransactionjson API

Returns the transaction information for a given transaction hash, if it is known to the network (i.e. mined or pending).

Request:

GET request - /gettransactionjson/<transactionHash>
  Params:
    <transactionHash> - hex string - A hash of the transaction, used to identify the transaction. Has no 0x prefix.

Response:

```
 {
   "txid" : <transactionHash>,
   "hash" : <transactionHash>,
   "version" : <version>,
   "size" : <size>,
   "vsize" : <vsize>,
   "locktime" : <locktime>,
   "vin" : [{
         "txid”: <inputTransactionHash>,
         "vout”: <vout>,
         "scriptSig”: {
               "asm": <scriptSigAsm>,
               "hex": <scriptSigAsm>,
             }
         "asm”: <asm>,
         "hex”: <hex>,
         "sequence”: <sequence>
        }],
   "vout" : [{
         "value”: <value>,
         "n”: <n>,
         "scriptPubKey”: {
               "asm”: <asm>,
               "hex”: <hex>,
               "reqSigs”: <reqSigs>,
               "type”: <type>,
               "addresses”: <addresses>
             }
        }],
   "hex" : <rawTransactionHex>,
   "blockhash" : <blockhash>,
   "confirmations" : <confirmations>,
   "time" : <time>,
   "blocktime" : <blocktime>
 }
```

<transactionHash> hex string – Hash of the transaction. Used to identify it.

<version> number – Transaction format version.

<size> number – length of the raw transaction data in bytes.

<vsize> number – length of the raw transaction data in bytes.

<locktime> number – No longer used for original purpose. Now typically set to 0. Once used in Bitcoin to indicate a block number only after which the transaction is to be considered valid.

<inputTransactionHash> hex string – Hash of the transaction that produced an output that is being used as input to this transaction.

<vout> number – The vins index as an output in the transaction that originated it (i.e. Output 1, output 2, etc)

<scriptSigAsm> string – Assembly representation of the scriptsig. Identifies two pieces of information to be put onto the execution stack: the transaction signature and the public key of the inputs address.

<scriptSigHex> hex string – Raw hex representation of the scriptsig.

<asm> string – Assembly representation of script information for the transaction output. Can some in 3 formats depending on the type of transaction output. Note that lower-case here represents a placeholder for real data while the non-lower-case represents literal values.

contract creation: 4 gaslimit gasprice bytecode OP_CREATE

sendtocontract: 4 gaslimit gasprice abi-encoded-data contractaddress OP_CALL

send: OP_DUP OP_HASH160 address OP_EQUALVERIFY OP_CHECKSIG

<hex> hex string – Raw hex representation of script information for the transaction output.

<sequence> hex string – No longer used for original purpose. Now typically set to maximum integer available in 4 bytes (0xFFFFFFFF). Originally used to override a transaction with a non-zero locktime.

<value> hex string – Value of FAB in the transaction output.

<n> hex string – Index of the output in the transaction.

<type> hex string – Script type.

<rawTransactionHex> hex string – Hexadecimal representation of the raw signed transaction data that was submitted to the blockchain network.

<blockHash> - hex string – Hash of the block the transaction is mined in. Used to identify that block.

<confirmations> hex string – Number confirmations of the transaction. 1 confirmation indicates that it was mined into a block. Every block mined after that increases the confirmations of the transaction by 1.

<time> hex string – UNIX time at which the block containing the transaction was mined.

<blocktime> number – UNIX time at which the block containing the transaction was mined.

gettransactionreceipt API

Returns the transaction receipt for a given transaction hash, if it is mined *and* it is a contract-related transaction such as contract creation or sendtocontract.

Request:

GET request - /gettransactionreceipt/<transactionHash>
  Params:
    <transactionHash> - hex string - A hash of the transaction, used to identify the transaction. Has no 0x prefix.

Response:

```
{
  "blockHash" : <blockHash>
  "blockNumber" : <blockNumber>,
  "transactionHash" : <transactionHash>,
  "transactionIndex" : <transactionIndex>,
  "from" : <from>,
  "to" : <to>,
  "cumulativeGasUsed" : <cumulativeGasUsed>,
  "gasUsed" : <gasUsed>,
  "contractAddress" : <contractAddress>,
  "excepted" : <excepted>
}
```

<blockHash> - hex string - A hash of the block containing the transaction, used to identify it. Has no 0x prefix.

<blockNumber> number – Number of the block containing the transaction.

<transactionHash> hex string – Hash of the transaction, used to identify it.

<transactionIndex> number – Position of the transaction within the block containing it.

<from> hex string – hexadecimal version of the sender's FAB address, with no 0x prefix. Used as the sender address in contracts.
<to> hex string – hexadecimal version of the FAB address associated with the recipient (in this context, a contract address), with no 0x prefix. For a contract creation, this may be a string of 0's.

<cumulativeGasUsed> - number – a measurement of the cost of execution based number and type of operations involved.

<gasUsed> - number – a measurement of the cost of execution based number and type of operations involved.

<contractAddress> hex string – The address of the contract called, or the address of the contracted created by the transaction.

<excepted> string – A short description of the error returned during execution of the contract related transaction, if one occurred. "None" if successful.

getblockbyhash or getblockbyheight API

Returns the transaction information for a given transaction hash, if it is known to the network (i.e. mined or pending).

Request:

GET request - /getblockbyhash/<blockHash>
OR
GET request - /getblockbyheight/<blockHeight>

Params:
<blockHash> - hex string - A hash of the block, used to identify it. Has no 0x prefix.
<blockHeight> - hex string – Number of the block in the blockchain.

Response:

```
{
  "hash" : <hash>,
  "confirmations" : <confirmations>,
  "strippedsize" : <strippedsize>,
  "size" : <size>,
  "weight" : <weight>,
  "height" : <height>,
  "version" : <version>,
  "versionHex" : <versionHex>,
  "merkleroot" : <merkleroot>,
  "hashStateRoot" : <hashStateRoot>,
  "hashUTXORoot" : <hashUTXORoot>,
  "tx" : {
        <tx>,
        ...
      },
  "time" : <time>,
  "medianTime" : <medianTime>,
  "nonce" : <nonce>,
  "bits" : <bits>,
  "difficulty" : <difficulty>,
  "chainWork" : <chainWork>,
  "previousBlockHash" : <previousBlockHash>,
  "nextBlockHash" : <nextBlockHash>
}
```

<hash> hex string – Hash of the block. Used to identify it.

<confirmations> hex string – Number confirmations of the block. 1 confirmation indicates that the block was mined. Every block mined after that increases the confirmations of the specific block by 1.
<strippedsize> number – size of the block in bytes not including witness data.

<size> number – size of the block in bytes.

<weight> number – size of the block in weight units. 1 weight unit represents 1/4,000,000th of the max block size.

<height> number – The block number of the block in the blockchain.

<version> number – Version number of the block.

<versionHex> hex string – Version number of the block, in hexadecimal.

<merkleroot> hex string – Hash of the merkle tree root. Represents a combined hash of every transaction in the blockchain.

<time> number – UNIX timestamp of the block was mined at.

<medianTime> number– Median of the last 11 block timestamps.

<nonce> hex string – nonce of the block.

<bits> string – Target threshold.

<difficulty> – Block difficulty. Determines the acceptable values for the next proof of work.

<chainWork> – Expected number of hashes expected to have produced the current chain.

<previousblockhash> – Hash of the preceding block in the chain.

<nextblockhash> – Hash of next block in the chain.