kanban smart contract development tools

This documentation provides information on various tools that can help you develop smart contracts on kanban. The choice of which tools to use will likely depend on your needs.

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 you take the time to set it up.

For writing simple smart contracts, such as a standard token contract (like the ERC20 standard), the Exchangily web-wallet can be used to deploy and interact with the contract, and the Exchangily API can be used for reading data for the contract.

For something that requires subscribing to events via web-sockets, or simply making direct RPC calls, it may be required to set or gain access to a kanban full-node. Note that for RPC calls in kanban, many ethereum-style rpc calls are available but rpc methods often use the prefix following prefix structure: kanban_methodName.

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 kanban network. Newer versions of solidity may work but some newer OPCODEs may not be supported on kanban. 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 a local (off-chain) test environment, and more. Either one is useful if you are trying to start writing a smart contract without having to spend too much time setting up your IDE.

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. Other useful development tools that are worth looking into for you smart contract development are Truffle, Ganache and Tenderly.

Deploying a contract

Once your source code is ready, you must compile your contract and retrieve an abi-encoded version of your smart contract. This process should get you:

ABI - a long JSON-based 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. NOTE: Some things may be labeled bytecode that are not one long hex string. Do not use them here.

constructor parameters - This depends on the contract code, but if a constructor requires parameters, they must be provided when deploying.

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 . After combining them into one string, you will have a long (or potentially extremely long) hex string that will populate the “data” field of the kanban transaction used to deploy it.

When deploying the contract, ensure that:

- The “data” field is populated by the abi-encoded contract code, as mentioned above.

- The “to” field is not included in the transaction. Unlike a typical transfer or senttocontract transaction, there is no recipient when we deploy a smart contract. Depending on what tools you are using to build the transaction, this field maybe need be blank or completely absent.

- The gas limit may need to be set sufficiently high that the contract may deploy successfully. This required gas limit can be much higher than a typical transaction, and may depend on the length of the contract, the complexity of the contract’s operations, or by the arguments passed to the constructor (if they initiate many operations). If you find that your deployment is failing, sometimes it bay be due to the gas limit being set too low.

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.

Sendtocontract Transactions

Calling a state-changing (i.e. not read-only) function on a contract requires a sendtocontract transaction. This type of transaction has the following properties:

- The “to” field is present and populated with the address of the contract to be called.

- 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.

- The gas limit is sufficient to cover the execution of the function. The required amount will often be smaller than for the original contract deployment, but the actual amount may vary quite a bit depending on the contract and function being called.

Calling a Contract

Calling a read-onlly function on a contract does not require a transaction, and is free. This requires a kind of pseudo-transaction that requires the following properties:

- The “to” field is present and populated with the address of the contract to be called.

- 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.

Exchangily Web Wallet Tools

Basics

To use the web wallet tools, you will need to set up a wallet. Go to exchangily.com and create a wallet if you haven't already done so. This can be done in section labeled "wallet". There is no need to login anywhere, as wallet data is stored locally. Make sure to keep your 12-word seed phrase safe so that you do not lose it, and secure so that another party can not find out what it is. Access to this seed phrase is equivalent to a backup of your private key. If it is lost or stolen, you can lose control of the funds in the wallet, and if a contract is programmed to have an "owner" then the privileges associated could be tied to a specific wallet address.

The wallet page provides information about various coin balances in kanban. Most important for development purposes is likely to be the "Kanban Gas" field. This value represents the native cryptocurrency of Kanban, which is used to pay transaction fees. (This includes regular send transactions, contract deployment transactions and sendtocontract transactions). This is measured in FAB, and originally comes from the Fabcoin blockchain via a cross-chain transaction. If you have a FAB balance on the FAB blockchain (in the FAB address of your exchangily wallet), you can use the "Add Gas" button to deposit the FAB into kanban. This will end up in the kanban address associated with that FAB address.

A note on FAB and kanban addresses:

These two types of addresses have the same private and public keys, therefore can be considered the same address, but they are represented in a hexadecimal format instead of base58. If you wish to convert from a FAB address to a kanban address, you can convert the FAB address from base58 to hexadecimal, then remove the first 2 and last 8 characters. Finally, add an 0x at the front.

Deploying A Smart Contract via the Web Wallet

Starting from the wallet page, navigate to the "Smart contract" section on the right. Select "Kanban Deploy". Here, there are two methods that can be used. If you already have abi-encoded string containing the compiled contract code and the encoded constructor arguments, you can simply place it in "bytecode" and leave the rest blank. If you have not combined them, and wish the tool to do it for you, populate the 3 fields with the appropriate values (ABI, bytecode without arguments, argument list). In either case, press "deploy". You may be asked for your password. This step should sign and submit a transaction to the network. After a few seconds, the transactionid shown now can be used to look up the transaction receipt (via api or rpc) to obtain the contract address and status (successful deployment should show a status of 0x1).

Making a Sendtocontract Transaction via the Web Wallet

Starting from the wallet page, navigate to the "Smart contract" section on the right. Select "Kanban Call".

Supply the address of the contract you wish to call in the "To" field.

For the value field, leave it at 0 unless the function you wish to call is designated as payable. This means that the contract is designed to receive onto the native coin of the chain (FAB in kanban a.k.a. "Kanban gas" in the web wallet).

If you already have an abi-encoded function call ready, put it in the "data" field and leave "ABI" and "Args" blank.

If you wish to have the tool do the abi-encoding for you, instead leave "data" blank, and supply the ABI (the large JSON object produced during compilation) in "ABI" and the argument list in "Args".

Press "Submit", entering your password if prompted. This will sign and submit your transaction to kanban network. If you click the transaction ID link that comes up, it should take you to a page with the transaction receipt info from the API. If this page seems empty, try waiting a bit and then refreshing, as the transaction may not have been mined yet.

Mnemonic Derivation

```
const Btc = require('bitcoinjs-lib');
const bs58 = require('bs58');
const ecc = require('tiny-secp256k1');
const BIP39 = require('bip39');
const { BIP32Factory } = require('bip32');
const BIP32 = BIP32Factory(ecc);

// This is an example of deriving keys and address from a mnemonic seed
phrase, for use with Fabcoin's ethereum-based child chain kanban.
// As a tradeoff between readability and portability to other languages,
only common crypto libraries are used.

const net = {
messagePrefix: '\x18Bitcoin Signed Message:\n',
bech32: 'bc',
bip32: {
public: 0x0488b21e,
private: 0x0488ade4,
},
pubKeyHash: 0x00,
scriptHash: 0x05,
wif: 0x80,
}

// generate bitcoin-esque base58 FAB address.
const seed = BIP39.mnemonicToSeedSync("<space separate seed phrase>");
let path = 'm/44\'/' + 1150 + '\'/0\'/' + 0 + '/' + 0;
let root = BIP32.fromSeed(seed, net);
let childNode = root.derivePath(path);
const privateKey = childNode.privateKey;
const { address } = Btc.payments.p2pkh({
pubkey: childNode.publicKey,
network: net
});
let bytes = bs58.decode(address)

// conversion of FAB address to Kanban address requires base58->hex then
removal of the first byte and final four bytes, and appended 0x
let hexAddressRaw= Buffer.from(bytes).toString('hex');
let kanbanAddress = `0x${hexAddressRaw.slice(2, 42)}`;

// arrangement of useful mnemonic-derived values.
let privateKeyHex = privateKey.toString('hex');
let privateKeyWIF = childNode.toWIF();
let publicKeyHex = childNode.publicKey.toString('hex');
let kanbanAddressHex = kanbanAddress;

// print it out if you wish to check your results. Be careful with private
keys. This is just a demonstration.
console.log(`privateKeyHex: ${privateKeyHex}`);
console.log(`privateKeyWIF: ${privateKeyWIF}`);
console.log(`publicKeyHex: ${publicKeyHex}`);
console.log(`kanbanAddressHex: ${kanbanAddressHex}`);
```

Recovering Sender Address from a Signed Kanban Transaction

The process of recovering a public key is similar but different to that of Ethereum. The public key is hashed in a way similar to bitcoin (rather than Ethereum), but produces 20 bit hexadecimal address similar to Ethereum. The following example uses popular Ethereum and cryptographic hash libraries.

```
const Ethjs = require('ethereumjs-tx');
const Common = require('ethereumjs-common');
const Util = require('ethereumjs-util');
const crypto = require('crypto');

const customCommon = Common.default.forCustomChain(
  'mainnet',
  {
    name: 'main',
    networkId: 211,
    chainId: 211
  },
  'byzantium'
)

// Use existing ethereum libraries for recovering ONLY the PUBLIC KEY
// Using these libraries address recovery functions will provide an
ethereum address and not a kanban address.
// Kanban-specific steps are outlined further down
let rawTx = '<example-hex-string-signed-raw-tx-goes-here>';
let transaction =
new Ethjs.Transaction(rawTx, { common: customCommon });

// Fetch uncompressed public key and convert from buffer to hex
string
const senderPubKeyUncompressed =
Util.bufferToHex(transaction.getSenderPublicKey());

// Convert from Uncompressed Public Key to Compressed Public Key
const isBufferEven = parseInt(senderPubKeyUncompressed.slice(-1), 16)
% 2 === 0;
const pubKeyPrefix = isBufferEven ? '02' : '03';
const senderPubKeyCompressed = pubKeyPrefix +
senderPubKeyUncompressed.slice(2, 66);

// Kanban-specific step: use bitcoin-style hashing (sha256 followed
by ripemd hash160)
// The result is given an 0x prefix and is now a 20 byte hexadecimal
address compatible with kanban
const sha256OfPubKey =
crypto.createHash('sha256').update(senderPubKeyCompressed,
'hex').digest();
const senderAddress = '0x' +
crypto.createHash('ripemd160').update(sha256OfPubKey).digest('hex');

console.log('senders recovered kanban address: ', senderAddress);
```

Signing a Transaction – Create Contract Transaction

See this example on the kanban explorer:
https://exchangily.com/explorer/transaction-detail/0x900de0ed59cee22902196c5fdf965869a2b418a9ab8ca73e44c6b46eeb6fd18d

```
const Ethjs = require('ethereumjs-tx');
const Common = require('ethereumjs-common');

// This is an example to demonstrate a signing process for contract
deployment on kanban blockchain.
// For the sake of simplicity, the example is a standard ERC20
contract.
const privateKey = new Buffer.from("<64-char-hex-string>", 'hex');

// Setup for kanban as custom chain
const customCommon = Common.default.forCustomChain(
  'mainnet',
  {
    name: 'main',
    networkId: 211,
    chainId: 211,
  },
  'byzantium',
)

let options = {
"from": "0x2eb00387c4bb98e2b425cf4110cb3eaaaa9ba049",
"value": 0,
"gas": 20000000,
"gasPrice": 50000000,
"input": "0x60806040...truncated-hex-string...00e8d4a51000",
"nonce": '0x4', // can get next nonce for address via
api.exchangily.com/kanban/getTransactionCount/:address or via rpc
method kanban_getTransactionCount
}

let transaction = new Ethjs.Transaction(options , { common:
customCommon });
transaction.sign(privateKey, customCommon);

const rawTransaction = `0x$
{transaction.serialize().toString('hex')}`;

console.log(`Raw Transaction: ${rawTransaction}`);

// Send this rawTransaction to API POST
api.exchangily.com/kanban/sendRawTransaction body:
{ signedTransactionData: rawTransaction }
// or RPC method kanban_sendRawTransaction
```

Signing a Transaction – SendToContract Transaction

See this example on the kanban explorer:
https://exchangily.com/explorer/tx-detail/0x7c170b59eb3f2e6509e8f8829ce68a6c8f5fe8154b9531f99a2430bf3e5c934d

```
const Ethjs = require('ethereumjs-tx');
const Common = require('ethereumjs-common');

// This is an example to demonstrate a signing process for a
sendtocontract transaction on kanban blockchain.
// Event tracking with getLogs is also included.
// For the sake of simplicity, the example is a standard ERC20
contract.
const privateKey = new Buffer.from("<64-char-hex-string>", 'hex');

// Setup for kanban as custom chain
const customCommon = Common.default.forCustomChain(
  'mainnet',
  {
    name: 'main',
    networkId: 211,
    chainId: 211,
  },
  'byzantium',
)

let options = {
"from": "0x2eb00387c4bb98e2b425cf4110cb3eaaaa9ba049",
"to": "0x9c959e3dc44c4c7b2469a4ada954b9e9fd9b1c56",
"value": 0,
"gas": 20000000,
"gasPrice": 50000000,
"input":
"0xa9059cbb0000000000000000000000006ced34107a815b7539397c2e3ac109d04f
0e094900000000000000000000000000000000000000000000000000000000000f424
0", // standard ERC20 transfer function abi
"nonce": '0x5', // can get next nonce for address via
api.exchangily.com/kanban/getTransactionCount/:address or via rpc
method kanban_getTransactionCount
}

let transaction = new Ethjs.Transaction(options , { common:
customCommon });
transaction.sign(privateKey, customCommon);

const rawTransaction = `0x$
{transaction.serialize().toString('hex')}`;

console.log(`Raw Transaction: ${rawTransaction}`);

// Send this rawTransaction to API POST
api.exchangily.com/kanban/sendRawTransaction body:
{ signedTransactionData: rawTransaction }
// or RPC method kanban_sendRawTransaction
```

Monitoring Logs – getLogs

We can also call getLogs to track an event triggered by the above transaction. The event is written into the contract as shown here:

```
event Transfer(address indexed from, address indexed to, uint
tokens);
```

Send an object like one below to

```
API POST api.exchangily.com/kanban/getLogs body: { paramsObject:
<the-params-object> }

or RPC method kanban_getLogs
```

Example:

```
All of the below fields are optional, but it is recommended to keep
the block range at a manageable size.

{
  "fromBlock":"0x2124B06",
  "toBlock": "0x2124B08",
  "address": "0x9c959e3dc44c4c7b2469a4ada954b9e9fd9b1c56",
  "topics":
["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"
,
"0x0000000000000000000000002eb00387c4bb98e2b425cf4110cb3eaaaa9ba049",
"0x0000000000000000000000006ced34107a815b7539397c2e3ac109d04f0e0949"]
}
```

For more details, check the sections of this document related to getLogs.

API Tools

The main api for kanban related requests can be found at https://api.exchangily.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 are a selection of routes provided the API that may be useful when developing, deploying, and managing smart contracts.

getBalance API

Returns the balance of the kanban blockchain's native coin, measured in FAB. It is used to pay for transaction fees on kanban. This FAB is originally deposited from the FAB blockchain into the FAB blockchain's SCAR contract.

Request:

```
GET request - /kanban/v2/getBalance/<address>
  Params:
    <address> hex string - a valid kanban address
```

Response:

```
  {
    "balance": <balance>
  }
```

<balance> - hex string - FAB balance of the address in kanban. The kanban network uses 18 decimals, so 1 x 10^{18} = 1 FAB. This differs from the fab blockchain, where there is a limit of 8 decimals. This just means that FAB in kanban can be divided into smaller pieces, and care must be taken if converting raw values.

Call API

Returns the balance of the kanban blockchain’s native coin, measured in FAB. It is used to pay for transaction fees on kanban. This FAB is originally deposited from the FAB blockchain into the FAB blockchain’s SCAR contract.

Request:

POST request - /kanban/call

```
  Params: N/A
  Body:
    {
      "transactionOptions": {
        "to": <to>
        "data": <data>
      },
      “defaultBlock”: <blockNumber>
    }
```

<to> hex string - address of the contract being called

<data> hex string - ABI-encoded string 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.

<blockNumber> (optional) string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending"

Response:

```
  {
    “data”: <result>
  }
```

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

sendRawTransaction API

Submits a transaction to the kanban network.

Request:

```
POST request - /kanban/call
  Params: N/A
  Body:
         {
                  "signedTransactionData": <rawTransaction>
         }
```

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

Response:

```
  {
    "transactionHash" : <transactionHash>
  }
```

<transactionHash> hex string - If successfully submitted to the kanban network, result will return the hash of the transaction submitted.

getTransactionReceipt API

Returns the transaction receipt for a given transaction hash, if it exists (if it has been mined).

Request:

GET request - /kanban/getTransactionReceipt/<transactionHash>
  Params:
    <transactionHash> - A hash of the transaction, used to identify the transaction.

  Body:
```
        {
                "signedTransactionData": <rawTransaction>
        }
```

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

Response:

```
  {
    "transactionHash" : <transactionHash>
  }
```

  <transactionHash> hex string - If successfully submitted to the kanban network, result will return the hash of the transaction submitted/.

getLogs API

Returns an array of event logs matching the given parameters.

Request:

GET request - /kanban/getTransactionReceipt/<transactionHash>
  Params:

<paramsObject> - An object structured as below.
  {
    "fromBlock" – (optional, default: "latest") string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.
     "toBlock" – (optional, default: "latest") string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.
    "address" – (optional) hex string or array of hex string – contract address or addresses that generated the logs.
    "topics" – (optional) array of hex string – array of 32 byte topics that identify certain events.
      Topics are order-dependent.
      Specifying *n* topics only forces the first *n* topics to match the provided list. A log with n+1 or more topics can still match a filter with *n* topics provided.
      Each topic can also be replaced by *null* as a placeholder
      Each topic can also be replaced by an array of topics that serves as an "OR" operator
      Example: the topic [null, [*a*, *b*]] means that the first topic can be anything, and the second topic must be topic *a* or topic *b.* There may be any number of additional topics after that, or none.
  }

Response:
  {   "logs" : <array of logs objects> }
log object - {
    "removed" – boolean – *true* if log removed by chain reorganization, otherwise false.
    "logIndex" – hex string or *null* – log position in block. *null* if pending log.
    "transactionIndex" – hex string or *null* – transaction position in block. *null* if pending log.
    "transactionHash" –hex string or *null* – hash of the transaction transaction the log was created from. *null* if pending log.
    "blockHash" –hex string or *null* – hash of the block the log was in. *null* if pending, or pending log.
    "blockNumber" - hex string or *null* – number of the block the log was in. *null* if pending, or pending log.
    "address" - hex string – address from which the transaction that triggered the log came from.
    "data" – hex string – contains one or more 32-bytes non-indexed arguments of the log.
    "topics" – array of hex string – 32-byte strings of indexed log arguments. The first topic is the hash of the signature of the event unless the even is *anonymous*.
    }

JSON RPC Tools

The JSON RPC tools for kanban are based off of the RPC format of Ethereum. If need something not listed below, try looking up an Ethereum RPC call here (https://github.com/ethereum/execution-apis) and replacing eth_methodName with kanban_methodName .

General Format

RPC request general format

```
{
  "jsonrpc":"2.0",
  "method":<method>,
  "params":<array of params>,
  "id": <id>
}
```

<method> string - the name of the method to be called. See method list.
<params> array - the parameter list for the function being called. See method list.
<id> string or number or null - The same id will be returned back in the RPC response. Can be used to differentiate responses when used in a combined data stream such as a web socket.

RPC response general format

```
{
   "jsonrpc": "2.0",
   "id": <id>,
   "result": <result>
}
```

Or

```
{
   "jsonrpc": "2.0",
   "id": <id>,
   "error": {
      "code": <error code>,
      "message": <error message>
   }
}
```

<id> string or number or null - the same id that was provided in the request.
<result> any - can be a variety of JSON data types, including object, array, string, number, etc. See method list for possible responses for a given method.
<error code> integer
<error message> string

Individual RPC Calls

kanban_getBlockByHash

Returns information about a block corresponding to the block hash provided.

params:

<block hash> hex string – A block hash, used to uniquely identify a particular block.
<show full transactions> - boolean - If false, the <transactions> array included in the response will be populated only by the hashes of transactions in the block. If true, will show more detailed informations.

Reponse:
  <result>:
   {
        _writeBack – hex string - (deprecated – irrelevant to kanban)

        crosschainRoot - hex string – a hash of the transactions found in crosschaintxs

        crosschaintxs – array of hex string – raw transactions submitted to third party

blockchains for cross chain withdrawal

        difficulty – hex string – (deprecated – irrelevant to kanban)

        extraData – hex string – Extra data included by the miner of the block.

        gasLimit – hex string - The total gas limit set by transactions in the block.

        gasUsed – hex string - The total gas used by transactions in the block.

        hash – hex string – hash of the block

        hashNoSignature – hash of the block header not including aggregate signature

        logsBloom – hex string – bloom filter of the logs of the block

        miner – hex string - address of the miner that mined the bloc

        mixHash - (deprecated – irrelevant to kanban)

        nextBlock – hex string – hash of the following block

        nonce – A value used to demonstrate proof-of work

        number – hex string – the number of the block in the blockchain

        parentHash – hex string - hash of the preceding block

        receiptsRoot – root of the receipts tree of the block

        round – number – the number of rounds it took to achieve consensus on the block

        sha3Uncles –hex string - combined hash of all uncles for a given parent

        parentHash – hex string - hash of the preceding block

        sha3Uncles –hex string - combined hash of all uncles for a given parent signature – hex string – aggregate signature created by pbft consensus

size – hex string - describes the size of the block information in bytes

stateRoot – hex string - The root hash of the Merkle trie

timestamp – hex string – unix timestamp of the time the block was mined

totalDifficulty – hex string - (deprecated – irrelevant to kanban)

transactions – array – structure varies. see above entry for <show full transactions>.

transactionsRoot – Root of the transactions trie of the block.

uncles – array – hashes of uncle blocks

}

kanban_getBlockByNumber

Returns information about a block corresponding to the block number provided.

params:

<block number> - hex string or one of 'latest', 'earliest', 'pending' – A block number in hex or one of 3 special labels.
<show full transactions> - boolean - If false, the <transactions> array included in the response will be populated only by the hashes of transactions in the block. If true, will show more detailed informations.

Reponse:
  <result>:
   {
        _writeBack – hex string - (deprecated – irrelevant to kanban)

        crosschainRoot - hex string – a hash of the transactions found in crosschaintxs

        crosschaintxs – array of hex string – raw transactions submitted to third party

blockchains for cross chain withdrawal

        difficulty – hex string – (deprecated – irrelevant to kanban)

        extraData – hex string – Extra data included by the miner of the block.

        gasLimit – hex string - The total gas limit set by transactions in the block.

        gasUsed – hex string - The total gas used by transactions in the block.

        hash – hex string – hash of the block

        hashNoSignature – hash of the block header not including aggregate signature

        logsBloom – hex string – bloom filter of the logs of the block

        miner – hex string - address of the miner that mined the bloc

        mixHash - (deprecated – irrelevant to kanban)

        nextBlock – hex string – hash of the following block

        nonce – A value used to demonstrate proof-of work

        number – hex string – the number of the block in the blockchain

        parentHash – hex string - hash of the preceding block

        receiptsRoot – root of the receipts tree of the block

        round – number – the number of rounds it took to achieve consensus on the block

        sha3Uncles –hex string - combined hash of all uncles for a given parent

        parentHash – hex string - hash of the preceding block

        sha3Uncles –hex string - combined hash of all uncles for a given parent

signature – hex string – aggregate signature created by pbft consensus

size – hex string - describes the size of the block information in bytes

stateRoot – hex string - The root hash of the Merkle trie

timestamp – hex string – unix timestamp of the time the block was mined

totalDifficulty – hex string - (deprecated – irrelevant to kanban)

transactions – array – structure varies. see above entry for <show full transactions>.

transactionsRoot – Root of the transactions trie of the block.

uncles – array – hashes of uncle blocks

}

kanban_syncing

Return the syncing status of the node. Response type can vary depending on whether it is syncing or not.

Params: (none)

Response:

```
  <result pattern A> - boolean – false

  <result pattern B> Object
  {
    startingBlock – string
    currentBlock – string
    highestBlock – string
    pulledStates – string
    knownStates – string
  }
```

kanban_coinbase

Returns the client node coinbase address.

Params: (none)

Response:
  <result> - hex string – The coinbase address.

kanban_accounts

Returns a list of addresses owned by the client node.

Params: (none)

Response:
  <result> - Array of hex string – The list of addresses.

kanban_blockNumber

Returns the block number of the most recently mined block.

Params: (none)

Response:

```
  <result> - Object
   {
     blockNumber -  string – The number of the most recent block in a decimal string.
     BlockNumberHex – hex string – The number of the most recent block in hex.
   }
```

kanban_call

A non-state changing call to a contract (read-only). Does not create a transaction on the blockchain and does not require a signature or transaction fee.

* fields marked *optional* are not usually required, but may occasionally be needed in special cases.

params:

<pseudoTransaction> - object -
{
    "to" – hex string – address of the smart contract to be called
    "data" - hex string – the abi-encoded data of the functional call to the smart contract
    "from" –(optional*) hex string – the address of the sender of the transaction.
    "gas" – (optional*) - hex string – gas limit of the function call.
    "gasPrice" -(optional*) hex string – gas price used for the transaction
    "value" -(optional*) hex string – value of the kanban's native currency (a form of FAB) sent to the contract as part of the function call.
}
<blockNumber> - string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.

Reponse:
<result> - hexadecimal string – abi-encoded return value from the function call. A return value of simply "0x" may (but does not necessarily) suggest that either the function or contract does not exist, or the call is otherwise invalid.

kanban_estimateGas

Generates an estimate of the gas required for the transaction provided. Every field is optional. If gas limit is not provided then it will use the block gas limit instead.

Params:

<pseudoTransaction> - object -
{
"to" – (optional) hex string – address of the smart contract to be called
"data" - (optional) hex string – the abi-encoded data of the functional call to the smart contract
"from" –(optional) hex string – the address of the sender of the transaction.
"gas" – (optional) - hex string – gas limit of the function call.
"gasPrice" -(optional) hex string – gas price used for the transaction
"value" -(optional) hex string – value of the kanban's native currency (a form of FAB) sent to the contract as part of the function call.
}

Reponse:
<result> - hexadecimal string – abi-encoded return value from the function call. A return value of simply "0x" may (but does not necessarily) suggest that either the function or contract does not exist, or the call is otherwise invalid.

kanban_gasPrice

Returns the current gas price.

Params: (none)

Response:
  <result> - hex string – The current gas price.

kanban_newFilter

Creates a log filter object to be notified of state changes.

Params:

<filter options> - object -
 {
  "fromBlock" – (optional, default: "latest") string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.
  "toBlock" – (optional, default: "latest") string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.
  "address" – (optional) hex string or array of hex string – contract address or addresses that generated the logs.
  "topics" – (optional) array of hex string – array of 32 byte topics that identify certain events.

Topics are order-dependent.

Specifying *n* topics only forces the first *n* topics to match the provided list. A log with n+1 or more topics can still match a filter with *n* topics provided.

Each topic can also be replaced by *null* as a placeholder

Each topic can also be replaced by an array of topics that serves as an "OR" operator

Example: the topic [null, [*a*, *b*]] means that the first topic can be anything, and the second topic must be topic *a* or topic *b.* There may be any number of additional topics after that, or none.
 }

Response:

<result> - hex string – a filter id

kanban_newBlockFilter

Creates a filter to notify when a new block appears. To check if the state has changed, call kanban_getFilterChanges.

Params: (none)

Response:
  <result> -hex string - a filter id

kanban_newPendingTransactionFilter

Creates a filter to notify when new pending transactions arrive. To see if the state has changed, call kanban_getFilterChanges.

Params: (none)

Response:
  <result> - hex string – a filter id.

kanban_uninstallFilter

Uninstalls a filter identified by its filter id.

Params:
  <filter id> - hex string – The filter id of the filter to be uninstalled.

Response:
  <result> - boolean – *true* if uninstalled successfully, otherwise *false*.

kanban_getFilterChanges

Polling method for a filter, which returns an array of logs since last poll.

Params:
  <filter id> - hex string – a filter id. See other filter related rpc calls for how to generate a filter id.

Response:

For filters generated by kanban_newBlockFilters:
   <result> - array of hex string – array of block hashes

For filters generated by kanban_newPendingTransactionFilter:
   <result> - array of hex string – array of transaction hashes

For filters generated by kanban_newFilter:
   <result> - array of log objects -
   {
      "removed" – boolean – *true* if log removed by chain reorganization, otherwise false.
      "logIndex" – hex string or *null* – log position in block. *null* if pending log.
      "transactionIndex" – hex string or *null* – transaction position in block. *null* if pending log.
      "transactionHash" –hex string or *null* – hash of the transaction transaction the log was created from. *null* if pending log.
      "blockHash" –hex string or *null* – hash of the block the log was in. *null* if pending, or pending log.
      "blockNumber" - hex string or *null* – number of the block the log was in. *null* if pending, or pending log.
      "address" - hex string – address from which the transaction that triggered the log came from.
      "data" – hex string – contains one or more 32-bytes non-indexed arguments of the log.
      "topics" – array of hex string – 32-byte strings of indexed log arguments. The first topic is the hash of the signature of the event unless the even is *anonymous*.
   }

kanban_getFilterLogs

Returns an array of all logs matching filter identified by filter id.

Params:
  <filter id> - hex string – The filter id of the filter to be uninstalled.

Response:
  <result> - see getFilterChanges

kanban_getLogs

Returns an array of all logs matching a given filter object.

Params:
<filter options> - object -
  {
    "fromBlock" – (optional, default: "latest") string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.
     "toBlock" – (optional, default: "latest") string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending". Select the block at which you wish to make the call.
    "address" – (optional) hex string or array of hex string – contract address or addresses that generated the logs.
    "topics" – (optional) array of hex string – array of 32 byte topics that identify certain events.
Topics are order-dependent.
Specifying *n* topics only forces the first *n* topics to match the provided list. A log with n+1 or more topics can still match a filter with *n* topics provided.
Each topic can also be replaced by *null* as a placeholder
Each topic can also be replaced by an array of topics that serves as an "OR" operator
Example: the topic [null, [*a*, *b*]] means that the first topic can be anything, and the second topic must be topic *a* or topic *b.* There may be any number of additional topics after that, or none.
  }

Response:
  <result> - <array of logs objects>

log object - {
"removed" – boolean – *true* if log removed by chain reorganization, otherwise false.
"logIndex" – hex string or *null* – log position in block. *null* if pending log.
"transactionIndex" – hex string or *null* – transaction position in block. *null* if pending log.
"transactionHash" –hex string or *null* – hash of the transaction transaction the log was created from. *null* if pending log.
"blockHash" –hex string or *null* – hash of the block the log was in. *null* if pending, or pending log.
"blockNumber" - hex string or *null* – number of the block the log was in. *null* if pending, or pending log.
"address" - hex string – address from which the transaction that triggered the log came from.
"data" – hex string – contains one or more 32-bytes non-indexed arguments of the log.
"topics" – array of hex string – 32-byte strings of indexed log arguments. The first topic is the hash of the signature of the event unless the even is *anonymous*.
}

kanban_getBalance

Returns a kanban address' balance in FAB. On kanban, 10^{18} smallest units = 1 FAB (whereas on FAB chain, 10^{8} smallest units = 1 FAB.

Params:
  <address> - hex string – an address whose balance needs to be retrieved.
  <blockNumber> string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending"

Response:
  <result> - hex string – balance of the given address, in smallest units

kanban_getStorageAt

Returns the value of data stored at specific position of a given contract address.

Params:
<address> - hex string – address of the storage
<position> - hex string - integer of the position in the storage
<blockNumber> string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending"

Response:
<result> - hex string – the abi-encoded value of the data at the storage position queried.

kanban_getTransactionCount

Returns a count of the number of transactions originating from an address,

Params:
<address> - hex string – an address whose balance needs to be retrieved.
<blockNumber> string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending"

Response:
<result> - hex string – number of transactions originating from the given address, before or at the supplied block number

kanban_getCode

Returns the code of a given contract address.

Params:
  <address> - hex string – address of the contract
  <blockNumber> string - hexadecimal representation of a blockNumber, or the strings "latest", "earliest", "pending"

Response:
 <result> - hex string – code of the contract.

kanban_sendRawTransaction

Params:
  <rawTransaction> hex string – a signed transaction encoded into a single hex string. For information on how to create and sign transactions, see

Response:
  <result> - hex string – If successfully submitted to the kanban network, result will return the hash of the transaction submitted. If unsuccessful, <result> will be replaced by <error>.

or

```
<error> Object
  {
    code – integer – error code
    message – string - error message
  }
```

kanban_getTransactionByHash

Returns information about a transaction selected using its transaction hash. This transaction may be mined or not mined, and simply known to the kanban node.

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

Response:
  <result> Object
  {
    hash – hex string - Hash of the transaction, used to identify it
    transactionIndex –hex string – Transaction's index; its position in the block.
    blockHash – hex string – Hash of the block, used to identify it.
    blockNumber – hex string – The number of the block in the blockchain.
    from – hex string – The address of the transaction's sender.
    to – hex string – The address of the transaction's recipient, if one exists.
    gas – hex string – The maximum amount of gas that the sender is willing to pay transaction costs for.
    gasPrice – hex string – The price the sender is paying for each unit of gas when executing the transaction.
    nonce – hex string – A number that determines the order of execution of the sender addresses transactions.
    value – hex string – The amount of native currency (FAB in kanban) being sent in the transaction, not including transaction fees.
    v - hex string – part of the transaction's signature
    r - hex string – part of the transaction's signature
    s - hex string – part of the transaction's signature
  }

kanban_getTransactionByBlockHashAndIndex

Returns information about a transaction selected using a combination of the containing block's hash and transaction's index within the block. This transaction may be mined or not mined, and simply known to the kanban node.

Params:
  <blockHash> hex string – Hash of the block, used to uniquely identify it.
  <transactionIndex> hexadecimal – Index of the transaction, identifying its position in the block.

Response:
  <result> Object
  {
    hash – hex string - Hash of the transaction, used to identify it
    transactionIndex –hex string – Transaction's index; its position in the block.
    blockHash – hex string – Hash of the block, used to identify it.
    blockNumber – hex string – The number of the block in the blockchain.
    from – hex string – The address of the transaction's sender.
    to – hex string – The address of the transaction's recipient, if one exists.
    gas – hex string – The maximum amount of gas that the sender is willing to pay transaction costs for.
    gasPrice – hex string – The price the sender is paying for each unit of gas when executing the transaction.
    nonce – hex string – A number that determines the order of execution of the sender addresses transactions.
    value – hex string – The amount of native currency (FAB in kanban) being sent in the transaction, not including transaction fees.
    v - hex string – part of the transaction's signature
    r - hex string – part of the transaction's signature
    s - hex string – part of the transaction's signature
  }

kanban_getTransactionByBlockNumberAndIndex

Returns information about a transaction selected using a combination of the containing block's number and transaction's index within the block. This transaction may be mined or not mined, and simply known to the kanban node.

Params:
  <blockNumber> hex string – Number of the block, used to identify it.
  <transactionIndex> hexadecimal – Index of the transaction, identifying its position in the block.

Response:
  <result> Object
  {
    hash – hex string - Hash of the transaction, used to identify it
    transactionIndex –hex string – Transaction's index; its position in the block.
    blockHash – hex string – Hash of the block, used to identify it.
    blockNumber – hex string – The number of the block in the blockchain.
    from – hex string – The address of the transaction's sender.
    to – hex string – The address of the transaction's recipient, if one exists.
    gas – hex string – The maximum amount of gas that the sender is willing to pay transaction costs for.
    gasPrice – hex string – The price the sender is paying for each unit of gas when executing the transaction.
    nonce – hex string – A number that determines the order of execution of the sender addresses transactions.
    value – hex string – The amount of native currency (FAB in kanban) being sent in the transaction, not including transaction fees.
    v - hex string – part of the transaction's signature
    r - hex string – part of the transaction's signature
    s - hex string – part of the transaction's signature
  }

kanban_getTransactionReceipt

Returns the receipt of a transaction selected using its transaction hash.

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

Response:

```
 <result> Object
 {
   transactionHash – hex string - Hash of the transaction, used to identify it
   transactionIndex –hex string – Transaction's index; its position in the block.
   blockHash – Hash of the block, used to identify it.
   blockNumber – The number of the block in the blockchain.
   from – The address of the transaction's sender.
   to – The address of the transaction's recipient, if one exists.
   cumulativeGasUsed – A measurement of the cost of execution of the transaction.
   gasUsed - A measurement of the cost of execution of the transaction.
   contractAddress – The address of the resulting contract, if one was created.
   logs – Array of Object – Information relating to the events emitted by the transaction.
      {
        address – hex string
        topics – Array of hex string – hashes of events triggered
        data - hex string – abi encoded string containing params
        blockNumber - hex string
        transactionHash - hex string
        transactionIndex - hex string
        blockHash - hex string
        logIndex - hex string
        removed - boolean
      }
   logsBloom -  hex string – A 2048 bit string containing a bloom filter for the logs.
   status – hex string – The success (if 0x1) or failure (if 0x0) of the transaction's execution.
 }
```