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 up a kanban full-node.

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 hexadecimal 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 hexadecimal 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) hexadecimal 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". 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.

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> hexadecimal string - a valid kanban address
```

Response:

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

<balance> - hexadecimal 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> hexadecimal string - address of the contract being called

<data> hexadecimal 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>: hexadecimal 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> hexadecimal 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> hexadecimal string - If successfully submitted to the kanban network, result will return the hash of the transaction submitted.