Hello, everyone!

In this article we will discuss some common techniques how to publish smart contracts and how to interact with them via Ethereum RPC API. The discussed API methods allow to resolve the following tasks:

1. Creation of an account
2. Creation of an appropriate smart contract
3. Receiving of smart contract information
4. Change of a smart contract state

**Contents**

* Some common remarks
* Wrapping of parameters and returned data
* Creation of an account and work with it
* Creation of a smart contract
  + Compilation of an initial smart contract code
  + Extraction of the code from a transaction
  + Computation of a contract publishing cost
  + Implementation of a contract publishing transaction
* Interaction with a smart contract
  + Creation of a contract with parameters
  + Identification of contract methods
  + Call for information request methods
  + Call for methods, changing a contract state

API description

* <https://github.com/ethereum/wiki/wiki/JSON-RPC>
* <https://github.com/ethereum/go-ethereum/wiki/Management-APIs>

**Some common remarks**

1. All the proposed steps are illustrated with the real data from the test net Rinkeby (at the time of writing)
2. You can track a state of transactions, accounts and smart contracts in Rinkeby via the website <https://rinkeby.etherscan.io/> (for a sub-net Ropsten, respectively, <https://ropsten.etherscan.io/>).

**Wrapping of parameters and returned data**

This part is of a general nature and that is why it is placed as a separate section. But you can postpone its investigation until you need to directly use some parameters or examine a smart contract response.

Incoming (parameters) or outcoming data package for a contract is formed according to the following principle:

* Data types, which are limited on the length (address, uint32, bytes32), are transmitted with alignment to 32-byte word (64-hex numbers)
* Data types, which are varying on the length (strings or arrays), are transmitted according to the following scheme:
  + In the object position in the list, we transmit the offset of the block and its data relatively to the beginning of the package (with alignment to 32-byte word)
  + A number of data items is transmitted in the first 32-byte word of the block
  + The data themselves are transmitted in the consequent 32-byte words

Let’s consider, for example, a block, where the following data are sent: address, string, uint32, address[] (a hexadecimal address relative to the beginning of the block is given in the beginning of every 32-byte sting for the convenience)

000:000000000000000000000000570f5d143ee469d12dc29bf8b3345fa5536476d9 020:0000000000000000000000000000000000000000000000000000000000000080 040:0000000000000000000000000000000000000000000000000000000000001234 060:00000000000000000000000000000000000000000000000000000000000000c0 080:0000000000000000000000000000000000000000000000000000000000000003 0a0:4e65770000000000000000000000000000000000000000000000000000000000 0c0:0000000000000000000000000000000000000000000000000000000000000002 0e0:000000000000000000000000aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 100:000000000000000000000000bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

* In the 000 string, we send the following address: 0х570f5d143ee469d12dc29bf8b3345fa5536476d9.
* In the 020 string, we send a link to the block, which describes a variable of a string type – 0x80 bytes from the beginning of block.
* In the 040 string, we send an integer 0x1234.
* In the 060 string, we send a link to a block, which describes an array address[] – 0xc0 bytes from the beginning of the block.
* In the 080 string, we send a symbols counter via a variable of the type string – 3.
* In the 0a0 string, we send symbols of the variable of the type string – a word New.
* In the 0c0 sting, we send a counter of the array address[] items – 2.
* In the 0e0 string, we send the first element of the address[] array - 0хaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.
* In the 100 string, we send the second element of the address[] array - 0хbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb.

**Pay attention!** The whole block is sent via a single coherent array.

**Creation of an account and work with it**

To create a new account, you can use personal\_newAccount method. The password is attached as a parameter, which later will be used to unblock an account:

{"jsonrpc":"2.0","method":"personal\_newAccount","params": ["PASSWORD"],"id":1}

An account ID is received as a response. In our case - 0xfbeda9914b78b58a0f0e810298f9d545f8379f8d.

{"jsonrpc":"2.0","id": 1,"result":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d"}

We will perform all the consequent steps via the following account - 0xfbeda9914b78b58a0f0e810298f9d545f8379f8d..

Now we have to put some amount of money to this account to pay for a transaction. As an outside person cannot take part in the mining within a test network Rinkeby, there is a special instrument described in <https://faucet.rinkeby.io/> to top an account.

To top an account, you need:

* Register in the github.com and create a new gist:

Изображение выглядит как текст

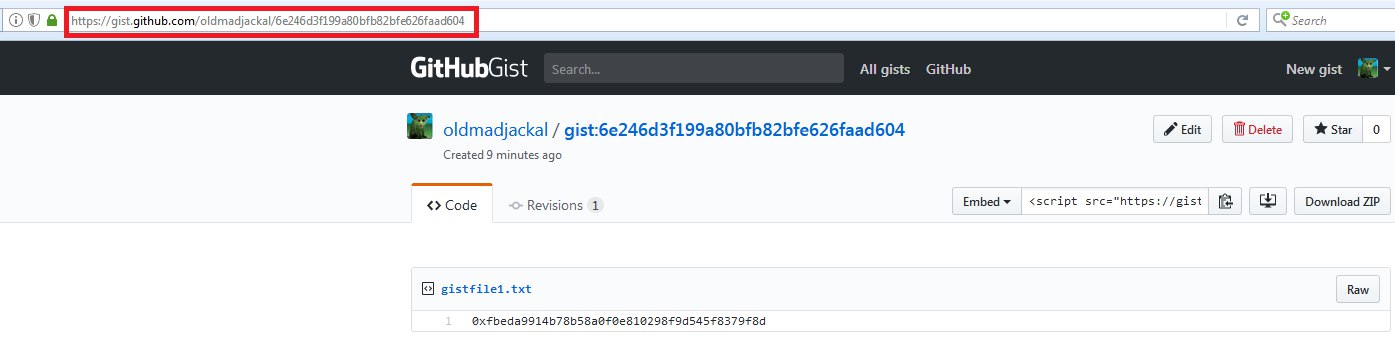
Автоматически созданное описание

* Put in an account ID and create a public gist (“Create public gist”):

Изображение выглядит как текст

Автоматически созданное описание

* Then we will have to use a URL of the gist – in the current case <https://gist.github.com/oldmadjackal/6e246d3f199a80bfb82bfe626faad604>:



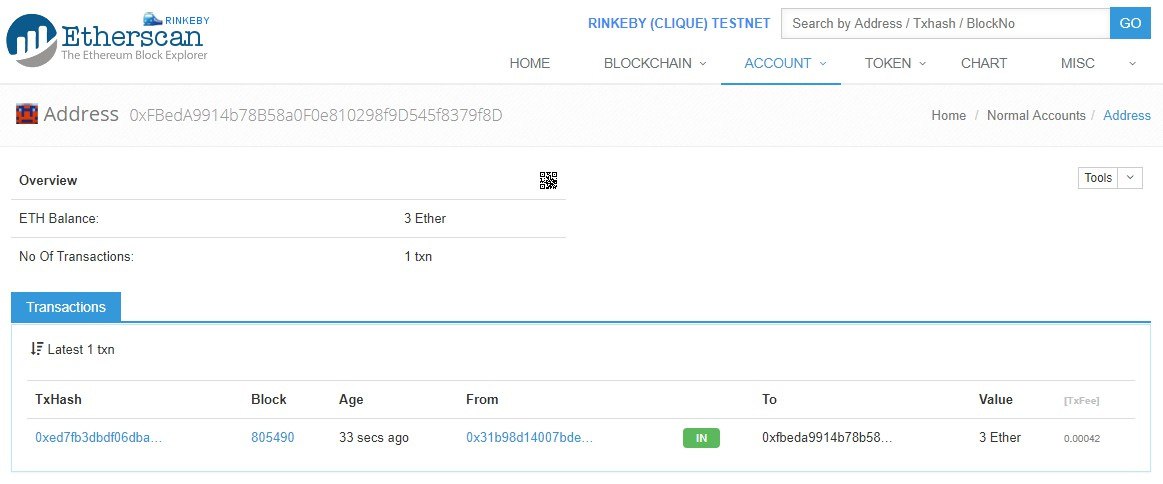
* Go to <https://faucet.rinkeby.io/>, enter a URL of the gist into a field and choose an amount of money to add to the account:

Изображение выглядит как текст

Автоматически созданное описание

If everything is okey, then a green approval message will appear. Transaction can take quite a long time – 5-10 minutes.

* Now we observe an account on EtherScan



* You can use the same gist for a repeating replenishment.

You have to **unlock an account** to use your funds from your account and perform any type of operations. To unlock your account, you can use personal\_unlockAccount method. You have to specify the following parameters: an account ID, a password and unlocking time in seconds:

{"jsonrpc":"2.0","method":"personal\_unlockAccount","params": ["0xfbeda9914b78b58a0f0e810298f9d 545f8379f8d", "PASSWORD", 600],"id":1}

{"jsonrpc":"2.0","id":1,"result":true}

**Creation of a smart contract**

Technically, a smart contract is created with a transaction which involves a contract bytecode and an parameters initialization block in the data field, if they are established in the smart contract.

You can obtain a contract bytecode to form a creation transaction in the following ways:

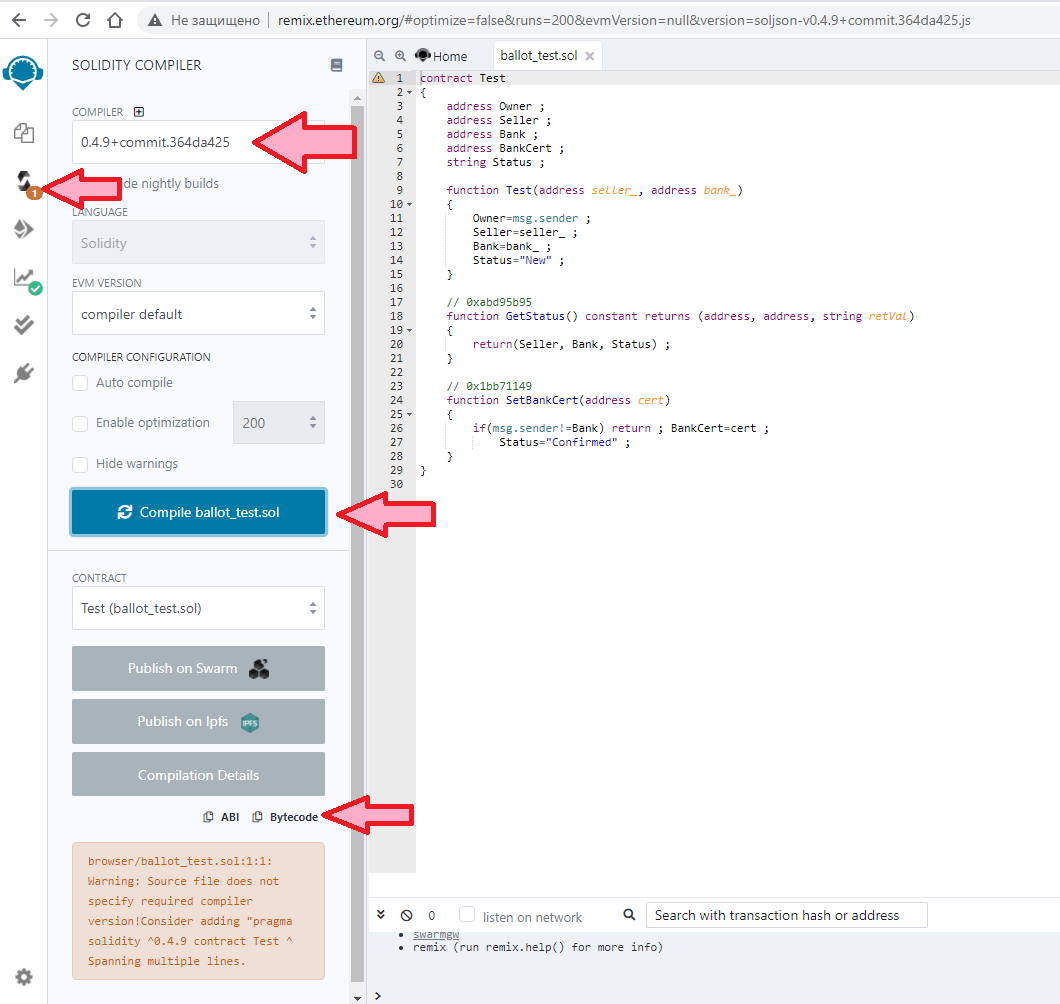
* A compilation from the initial smart contract text
* An extraction of a similar smart contract from another creation transaction

The second way can be used to store a template for a contract creation directly in the blockchain – then it is enough to know a transaction ID to extract and “clone” it.

Technically, you can extract a transaction code directly via an ID of an existing smart contract (method eth\_getCode). But it is **PROHIBITED**, as it produces a bytecode which was subjected to a special processing during a smart contract creation (particularly, due to this, some initialization blocks are removed).

***Compilation of an initial smart contract code***

I use Remix - <https://remix.ethereum.org/> - to compile an source code via Solidity.



Enter a smart contract text. If there are no errors, there will be a bytecode in the Bytecode field. And in the current case we have:

 e57bdb35f0a3cdbfe8f2ece20d9a3eaa0029

***Extraction of the code from a transaction***

You can use eth\_getTransactionByHash method to extract code from transactions. In our example, a “cloned” smart contract was created via a transaction 0xc4d20bb8f9eede45968fc6bc850397e92b9f263eeb11200615cc08562d46c2e7.

{"jsonrpc":"2.0","method":"eth\_getTransactionByHash","params": ["0xc4d20bb8f9eede45968fc6bc850397e92b9f263eeb11200615cc08562d46c2e7"],"id":1}

An **input** tag includes a contract bytecode.

{"jsonrpc":"2.0","id":1,"result":{"blockHash":"0x731f202b0232de8c474c71677b29868f65802c068d0de31b17bec09f3e31144c","blockNumber":"0xbad3f","from":"0x17eafa57fd812968d90aecc5a51e330e2e1c11a3","gas":"0x31b2e","gasPrice":"0x9c7652400","hash":"0xc4d20bb8f9eede45968fc6bc850397e92b9f263eeb11200615cc08562d46c2e7",  
"input":"",  
"nonce":"0x30","to":null,"transactionIndex":"0x2","value":"0x0","v":"0x2c","r":"0x577865931aac644a3eefb83b59344caeab9e2970cfeb1bef02837b1bb4bccca0","s":"0x48013e0c5ca3bff66fa7e19ebbcd9eaed63bc0a45004041c310f9fc4dfc5e5e8"}}

Pay attention: if a smart contract constructor suggests a parameters usage, tag input will include not only a bytecode, but also parameters values, following a bytecode. You can use zero values during a smart contract creation for a consequent cloning.

***Computation of a contract publishing cost***

You can use eth\_estimateGas method to compute a contract publishing value. In the parameters of this method, you have to specify an account number (tag “from”), from which a smart contract will be created, and also a contract bytecode (tag “data”). There will be specified a sufficient amount of Gas in the response.

If a smart contract constructor suggests a parameters usage, they should be included into a request in the “data” tag right after a bytecode. Otherwise, computation of Gas will be incorrect.

{"jsonrpc":"2.0","method":"eth\_estimateGas","params":[{"from":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d",  
"data":"0x6060604052341561000f57600080fd5b5b6040805190810160405280600381526020017f4e6 "  
}], "id":1}

{"jsonrpc":"2.0","id":1,"result":"0x31b2e"}

Pay attention, in some cases smart contracts include some semantic (not syntax) mistakes, which can be missed by the compiler and cause the unserviceability of a contract.

***Implementation of a contract publishing transaction***

You can use eth\_sendTransaction method to publish a contract. The following items are sent as parameters:

* An account number, from which a smart contract is created (tag “from”)
* A cost of publishing in Gas (tag “gas”, which is obtained from the previous point)
* A contract bytecode with an attached block of parameters of the constructor (tag “data” has to be totally similar to the one used in the previous point)

{"jsonrpc":"2.0","method":"eth\_sendTransaction","params":[{"from":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d","gas":"0x31b2e",  
"data":" 2301a980e57bdb35f0a3cdbfe8f2ece20d9a3eaa0029" }], "id":1}

We receive a transaction number as a response:

{"jsonrpc":"2.0","id":1,"result":"0x26cd429a43bc2f706f206fa6a536374cc7bf0e5090f0ed9b8f30ded71 73529f5"}

Or an error message:

{"jsonrpc":"2.0","id":1,"error":{"code":-32000,"message":"authentication needed: password or unlock"}}

Now you have to wait for a transaction commitment and get its result – whether a contract is created or not. You can use eth\_getTransactionReceipt to do this:

{"jsonrpc":"2.0","method":"eth\_getTransactionReceipt","params": ["0x26cd429a43bc2f706f206fa6a536374cc7bf0e5090f0ed9b8f30ded7173529f5"],"id":1}

While a transaction is in the “waiting list”(Pending Txn), there will be a following answer:

{"jsonrpc":"2.0","id":1,"result":null}

After a transaction commitment we will receive a full “receipt”, a “contractAddress” tag of which will include an address of a created smart contract:

{"jsonrpc":"2.0","id":1,"result":{"blockHash":"0x3afdc600435caebebb91497f01372c3ad6ac712c37fe9b1028445d8b41a58fca","blockNumber":"0xc4b16",  
"contractAddress":"0x2af49c8a413ea4b66ca8fd872befa9d1c8d22562",  
"cumulativeGasUsed":"0x31b2d","from":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d","gasUsed":"0x31b2d","logs":[],"logsBloom":"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000","root":"0x47b6297fef6ba21c45ea845a6fba19ddb144d957f600cfe7230e05f0f711ac12","to":null,"transactionHash":"0x26cd429a43bc2f706f206fa6a536374cc7bf0e5090f0ed9b8f30ded7173529f5","transactionIndex":"0x0"}}

If the receipt is not received in 5 minutes, there are either some problems in the net, or your node has not sent transaction within the net. To understand an underlying cause, you have to examine a Pending Txn queue via etherscan.io (<https://rinkeby.etherscan.io/txsPending>). If there is no transaction, it means that you have to restart an Ethereum client and repeat a publication again.

Now you have to check, whether a smart contract is created correctly. To do this, you can use eth\_getCode method – a receiving of smart contract code based via its address:

{"jsonrpc":"2.0","method":"eth\_getCode","params": ["0x2af49c8a413ea4b66ca8fd872befa9d1c8d2256 2", "latest" ],"id":1}

If a smart contract is created incorrectly, there will be the following response:

{"jsonrpc":"2.0","id":1,"result":"0x"}

If there are some data, different from 0x, then a smart contract is created successfully.

**Interaction with a smart contract**

We will use the following smart contract sample to demonstrate smart contract interaction:

pragma solidity ^0.4.9;

contract Test

{

address Owner ;

address Seller ;

address Bank ;

address BankCert ;

string Status ;

function Test(address seller\_, address bank\_)

{

Owner=msg.sender ;

Seller=seller\_ ;

Bank=bank\_ ;

Status="New" ;

}

// 0xabd95b95

function GetStatus() constant returns (address, address, string retVal)

{

return(Seller, Bank, Status) ;

}

// 0x1bb71149

function SetBankCert(address cert)

{

if(msg.sender!=Bank) return ; BankCert=cert ;

Status="Confirmed" ;

}

}

It receives a seller address (seller\_) and a bank address (bank\_) during a smart contract creation (constructor – a Test function). GetStatus method returns an address of a seller, an address of a bank, and a current contract status. SetBankCert method is used to save some numerical ID with transmission to a “Confirmed” status in the contract.

Contract bytecode:

 9516dd1860029

***Creation of a contract with parameters***

If a smart contract constructor uses parameters (as in or demo-sample), they have to be “wrapped” in accordance with “Wrapping of parameters and returned data” description and attached to the tail of a contract bytecode.

In our case, transmission of seller’s address 0x794ce6de39fa2d274438cc1692db04dfb5bea836 and bank’s address 0xfbeda9914b78b58a0f0e810298f9d545f8379f8d during a smart contract creation will look as follows (bytecode ends up with 0029):

60606040...94000029000000000000000000000000794ce6de39fa2d274438cc1692db04dfb5bea8360 00000000000000000000000fbeda9914b78b58a0f0e810298f9d545f8379f8d

Pay attention once more: sGas computation should be performed for a whole block Bytecode + Parameters. Otherwise, contract will not be created, or it will not work.

Our test demo-contract was created on the address 0x3d20e579f5befdc7d3f589adb6155f684d9a751c.

***Identification of contract methods***

The first four bytes (eight hexadecimal numbers) from a method description hash are used to identify a smart contract method.

For example, GetStatus() will be used as a description for GetStatus contract method. For a SetBankCert, it will be a SetBankCert(address). There should be a special emphasis that there are no **spaces** in the description – there have been some bad accidents ☹.

You can use web3\_sha3 method to define a hash, and in such case a string value should be given as a hexadecimal number (it will be 0x4765745374617475732829 for GetStatus()):

{"jsonrpc":"2.0","method":"web3\_sha3","params":["0x4765745374617475732829"],"id":1}

{"jsonrpc":"2.0","id": 1,"result":"0xabd95b950242a279866243fa2b8fec5adddf6560d4e1b4f8745cfe7b5 7786865"}

Respectively, GetStatus method will have 0xabd95b95 as an identifier, and SetBankCert method will have 0x1bb71149 as an identifier.

***Call for information request methods***

You can use API eth\_call to call for methods, which are not related to a contract state change (for example, to receive information about its current status).

A call for a method can have a following structure:

{"jsonrpc":"2.0","method":"eth\_call","params":[{"to":<Contract address>, "data":<Call data>},"latest"],"id":1}

A block “call data” can be formed in a following way:

<Method ID><Parameters data>

where <Parameters data> are formed accordingly to the section “Wrapping of parameters and returned data”.

If a method does not suggest any parameters, then a <Call data> block includes only a method ID.

For example, the following request is used to call for a GetStatus method of the demo-contract Test:

{"jsonrpc":"2.0","method":"eth\_call","params":[{"to":"0x3d20e579f5befdc7d3f589adb6155f684d9a751c", "data":"0xabd95b95"}, "latest"],"id":1}

and the response will be as follows:

{"jsonrpc":"2.0","id":1,"result":"0x000000000000000000000000794ce6de39fa2d274438cc1692db04dfb5bea836000000000000000000000000fbeda9914b78b58a0f0e810298f9d545f8379f8d000000000000000000000000000000000000000000000000000000000000006000000000000000000000000000000000000000000000000000000000000000034e65770000000000000000000000000000000000000000000000000000000000"}

Let’s examine a received response in accordance with the rules from a “Wrapping of parameters and return data” section. We will also take into consideration a description of GetStatus method – a function GetStatus() constant returns (address, address, string retVal).

As a matter of convenience, we will resolve a response into make a 32-byte:

000:000000000000000000000000794ce6de39fa2d274438cc1692db04dfb5bea836 020:000000000000000000000000fbeda9914b78b58a0f0e810298f9d545f8379f8d 040:0000000000000000000000000000000000000000000000000000000000000060 060:0000000000000000000000000000000000000000000000000000000000000003 080:4e65770000000000000000000000000000000000000000000000000000000000

On the basis of the description, we expect to receive following variables: address, address, string. So:

* A “000” string includes a Seller’s address (of a type address) - 0x794ce6de39fa2d274438cc1692db04dfb5bea836
* A “020” sting includes a Seller’s address (of a type address) - 0xfbeda9914b78b58a0f0e810298f9d545f8379f8d
* A “040” string includes a link to a block of a contract state description (of a type string) – a block starts with address “060”
* A “060”string includes a counter of symbols in the contract status string – 3 symbols
* A “080”string includes symbols of a contract status in the hexadecimal coding - New

***Call for methods, changing a contract state***

You can use API eth\_sendTransaction method to call for methods, which can change a contract state.

A call for a method has the following structure:

{"jsonrpc":"2.0","method":"eth\_sendTransaction","params": [{ "from":<Initializer address>, "to":<Contract address>, "gas":<Implementation cost>, "data":<Call data> }], "id":1}

<Initializer address> should have a balance, sufficient to pay for a <Implementation cost>. Moreover, it should be noted that a contract may include inner conditions about the control over <Initializer address>, as, for example, in SetBankCert method of out demo-contract. <Call data> block is formed as follows:

<Method ID><Call data>,

where <Parameters data> as formed in accordance with “Wrapping of parameters and returned data” section. If a method does not suggest any parameters, then a <Call data> block includes only a method ID.

For example, in case of calling to a SetBankCert method ("0хf7b0f8870a5596a7b57dd3e035550aeb5af16607") of a demo-contract, <Call data> will look as follows:

0x1bb71149000000000000000000000000f7b0f8870a5596a7b57dd3e035550aeb5af16607.

You can use eth\_estimateGas method to define an implementation cost and also to create a smart contract. The parameters, sent to this method, should be the same as the ones sent to eth\_sendTransaction method.

{"jsonrpc":"2.0","method":"eth\_estimateGas","params":[{"from":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d", "to":"0x3d20e579f5befdc7d3f589adb6155f684d9a751c", "data":"0x1bb71149000000000000000000000000f7b0f8870a5596a7b57dd3e035550aeb5af16607"}],"id":1}{"jsonrpc":"2.0","id":1,"result":"0xd312"}

In our experience, if a requested method includes a transaction call to other smart contracts, then a Gas sum can be computed incorrectly and a transaction will not be committed. Consequently, I strongly recommend you specify inherently higher amount of Gas, as, in theory, any excess will not be used. In similar cases, I personally specify an amount of Gas, which is close to the maximum – 0x200000.

Then we call to eth\_sendTransaction method:

{"jsonrpc":"2.0","method":"eth\_sendTransaction","params":[{"from":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d","to":"0x3d20e579f5befdc7d3f589adb6155f684d9a751c","gas":"0xd312","data":"0x1bb71149000000000000000000000000f7b0f8870a5596a7b57dd3e035550aeb5af16607"}],"id":1}

and then we receive a transaction ID as a response:

{"jsonrpc":"2.0","id":1,"result":"0xe55c9fe8f816f5730053fc491ea27acfd83c615b6623d06f25fb281fea 750f3c"}

Similar to a smart contract creation, we should wait until a transaction is submitted when requesting a receipt (eth\_getTransactionReceipt):

{"jsonrpc":"2.0","method":"eth\_getTransactionReceipt","params":["0xe55c9fe8f816f5730053fc491ea27acfd83c615b6623d06f25fb281fea750f3c"],"id":1}

As soon as a receipt is delivered, a transaction is committed:

{"jsonrpc":"2.0","id":1,"result":{"blockHash":"0xd8bb4a0b0ca3a598a69786f2f40876d547a672044fc8d961ec22da60606fa2fb" , "blockNumber":"0xc4b6f", "contractAddress":null, "cumulativeGasUsed":"0x459e5", "from":"0xfbeda9914b78b58a0f0e810298f9d545f8379f8d", "gasUsed":"0xd311","logs":[],"logsBloom":"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000","root":"0x75f6823221a4f0e4f80dfa6c7ccfc961c7522833aac959d6a398537730a62041","to":"0x3d20e579f5befdc7d3f589adb6155f684d9a751c","transactionHash":"0xe55c9fe8f816f5730053fc491ea27acfd83c615b6623d06f25fb281fea750f3c", "transactionIndex":"0x1"}}

However, transaction implementation itself does not mean that a smart contract has shifted to a necessary state. Moreover, inner transaction calls to other smart contracts can also not be performed.

In my opinion, the most reliable way to verify whether everything works properly is to request a smart contract state and make sure that everything has changed “as it is supposed to”.

000:000000000000000000000000794ce6de39fa2d274438cc1692db04dfb5bea836 020:000000000000000000000000fbeda9914b78b58a0f0e810298f9d545f8379f8d 040:0000000000000000000000000000000000000000000000000000000000000060 050:0000000000000000000000000000000000000000000000000000000000000009 060:436f6e6669726d65640000000000000000000000000000000000000000000000

Having examined a response, we can see that a status has changed to “Confirmed”.

**Summary**

In the current manual we have discussed implementation of different tasks, related to smart contract publishing and interaction with them via RCP API blockchain platform Ethereum. We have figured out how to create an account, create a particular smart contract, receive information about it and change its state. We have also revealed how to interact with this smart contract.

Please don't hesitate to contact me if you have any questions; I would be happy to reply to any of your messages.