Quantum (Qtum) Blockchain Developer Tutorial - Hello World!

cryptominder (38) (/@cryptominder)in **#qtum (/trending/qtum)** • 3 years ago

Introduction

In my previous post (https://steemit.com/qtum/@cryptominder/qtum-blockchain-development-environment-setup), I described how to set up a 3-node **regtest** Quantum (Qtum) blockchain environment using Docker. Hopefully you have this up and running -- since we'll be using it in this tutorial.

As a first introduction to developing a smart contract on the Quantum blockchain, we'll be re-creating the **Ethereum Greeter** tutorial at https://ethereum.org/greeter . We'll re-use the same Solidity contract, but we'll be creating and invoking the contract using the Quantum command-line tool (i.e. `qtum-cli`).

After completing this initial tutorial, you will hopefully gain a better understanding of the Quantum blockchain and be inspired to develop novel **decentralized applications.**

Before We Start... Why Quantum (Qtum) instead of Ethereum?

This is a good question, which also has a very good answer.

As you know, Quantum (a.k.a. Qtum) has leveraged and also evolved the strengths of other blockchain projects, namely:

- **Bitcoin:** For its UTXO model (and the security this provides), as well as support for Simplified Payment Verification (SPV).
- **Ethereum:** For its smart contract execution virtual machine (i.e. the EVM).
- **Blackcoin:** For its Proof-of-Stake (POS) model, over which Quantum also added some improvements.

The Quantum **Account Abstraction Layer** innovation enables the UTXO (address) model to interoperate with the Ethereum (account) model. In addition to the Ethereum VM, it was recently announced that Qtum is working on supporting other VMs.

Support for Simplified Payment Verification (SPV) allows Quantum clients to run on devices where it is not practical to download and persist full blocks of the blockchain (e.g. due to bandwidth costs and/or storage constraints). As such, SPV support enables Quantum apps to run on your iOS (e.g. iPhone / iPad) or Android device -- and even on a satellite. While Quantum supports SPV, it should be noted that this is not the only part of their strategy (i.e. they are not solely focused on the mobile market).

Quantum's support for Proof-of-Stake makes it possible to mint new blocks using lower-power devices such as the Raspberry Pi (https://steemit.com/qtum/@cryptominder/qtum-staking-tutorial-using-qtumd-on-a-raspberry-pi-3). This is much more environmentally friendly than Proof-of-Work as used by Bitcoin and (still today) Ethereum.

Also, to avoid disruptive forks, Quantum supports a Decentralized Governance Protocol to safely modify blockchain parameters.

For the reasons given above (and others, such as their stellar team), the Quantum blockchain offers several key advantages over Ethereum.

Assumptions

In addition to my earlier assumptions (https://steemit.com/qtum/@cryptominder/qtum-blockchain-development-environment-setup), I will be making these additional assumptions:

- You have Docker installed and a Qtum `regtest` environment (https://steemit.com/qtum/@cryptominder/qtum-blockchain-development-environment-setup) running.
- You are running the commands below from the directory containing the **config files** (e.g. node1_qtumd.conf) and the **datadir** sub-directories (e.g. node1_data).
- If desired, you are able to create parameterized batch/shell scripts for the (rather long) Docker commands below.

When I use the `${PWD}` environment variable (below), it is replaced by the current directory under Linux and macOS/OSX. On **Windows**, this is typically equivalent to `%cd%`.

With the `-v` parameter to `docker run`, it's important that an **absolute** path is given.

Step 1 - Are you ready?

Before we get started, let's make sure we have things in order. Let's first check that our 3 **qtumd** nodes are running:

```
$ docker ps -f name=qtumd
```

The command above should return 3 running Docker containers (i.e. `qtumd_node1`, `qtumd_node2`, and `qtumd_node3`).

Next, let's make sure we're at block 600 or higher, and that we have a positive balance **in at least 1 wallet**:

```
$ docker run -i --network container:qtumd_node1 -v
```

```
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli getinfo
```

The above should return something like:


```
{
  "version": 140301,
  "protocolversion": 70016,
  "walletversion": 130000,
  "balance": 2000000.00000000,
  "stake": 0.00000000,
  "blocks": 600,
  "timeoffset": 0,
  "connections": 2,
  "proxy": "",
  "difficulty": {
    "proof-of-work": 4.656542373906925e-10,
    "proof-of-stake": 4.656542373906925e-10
  },
  "testnet": false,
  "moneysupply": 12000000,
  "keypoololdest": 1507588445,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00400000,
  "errors": ""
}
```


Notice that `blocks` is at `600`, and `balance` is `2000000.00000000`. The `2000000.00000000` balance gives us many QTUM coins to play with…

If you run the same `getinfo` command on the other nodes (e.g. qtumd_node2 and qtumd_node3), the `block` number should be the same or higher (depending on how long you waited), but the `balance` will likely be `0`, e.g.:

```
docker run -i --network container:qtumd_node2 -v
```

${PWD}/node2_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node2_data:/data cryptominder/qtum:latest qtum-cli getinfo

can return:


```
{
  "version": 140301,
  "protocolversion": 70016,
  "walletversion": 130000,
  "balance": 0.00000000,
  "stake": 0.00000000,
  "blocks": 605,
  "timeoffset": 0,
  "connections": 2,
  "proxy": "",
  "difficulty": {
    "proof-of-work": 4.656542373906925e-10,
    "proof-of-stake": 4.656542373906925e-10
  },
  "testnet": false,
  "moneysupply": 12100000,
  "keypoololdest": 1507588475,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00400000,
  "errors": ""
}
```


This is fine.

Now, let's check if there are any existing smart contracts… run the following:

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli listcontracts
```

You should see 5 contracts listed on the blockchain:

```
{
  "0000000000000000000000000000000000000083": 0.00000000,
  "0000000000000000000000000000000000000080": 0.00000000,
  "0000000000000000000000000000000000000081": 0.00000000,
  "0000000000000000000000000000000000000082": 0.00000000,
  "0000000000000000000000000000000000000084": 0.00000000
}
```

None of these contracts were deployed by us -- ignore them (at least for this tutorial).

Lastly, we'll be using `qtumd_node1` as the node on which we will usually invoke `qtum-cli`, but I encourage you to try the other nodes too.

Step 2 - Pulling a few tools: solc and ethabi

For the Solidity compiler, we will be using the **solc** Docker image as described in the Solidity documentation . Pull the image using:

```
$ docker pull ethereum/solc
```

Check that it works with:

```
$ docker run --rm -v ${PWD}:/solidity ethereum/solc:stable --version
```

The command above should output:

```
solc, the solidity compiler commandline interface
Version: 0.4.17+commit.bdeb9e52.Linux.g++
```

Notice that we used `-v ${PWD}:/solidity`. The `/solidity` directory within the image is set to the **working directory**, so that's why we're mapping our current (absolute path) there.

You can run the following command to get help with **solc**:

```
$ docker run --rm -v ${PWD}:/solidity ethereum/solc:stable --help
```

We'll also need the ethabi command-line tool . I created a Docker image for it at https://hub.docker.com/r/cryptominder/ethabi/ .

Pull the **ethabi** image using:

```
$ docker pull cryptominder/ethabi
```

Check that it works with:

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest --help
```

The above should give you:

```
Ethereum ABI coder.
  Copyright 2016-2017 Parity Technologies (UK) Limited

Usage:
    ethabi encode function <abi-path> <function-name> [-p <param>]...
    ethabi encode params [-v <type> <param>]... [-l | --lenient]
    ethabi decode function <abi-path> <function-name> <data>
    ethabi decode params [-t <type>]... <data>
    ethabi decode log <abi-path> <event-name> [-l <topic>]... <data>
    ethabi -h | --help

Options:
    -h, --help          Display this message and exit.
    -l, --lenient       Allow short representation of input params.

Commands:
    encode              Encode ABI call.
    decode              Decode ABI call result.
    function            Load function from json ABI file.
    params              Specify types of input params inline.
    log                 Decode event log.
```

Notice again that we used the `-v ${PWD}:/ethabi` option in `docker run`. The `/ethabi` directory within the image is set to the **working directory**, so that's why we're mapping our current (absolute path) there.

Ok, we're now ready to really get started...

Step 3 - Compile the smart contract

We'll re-use the exact same smart contract that is shown at https://ethereum.org/greeter .

Create the following file, named: `helloworld.sol`:

```
contract mortal {
    /* Define variable owner of the type address */
    address owner;

    /* This function is executed at initialization and sets the owner
    function mortal() { owner = msg.sender; }

    /* Function to recover the funds on the contract */
    function kill() { if (msg.sender == owner) selfdestruct(owner); }
}

contract greeter is mortal {
    /* Define variable greeting of the type string */
    string greeting;

    /* This runs when the contract is executed */
    function greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* Main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
```

Compile the Solidity code by running:

```
$ docker run --rm -v ${PWD}:/solidity ethereum/solc:stable --optimize
--bin --abi --hashes -o /solidity --overwrite /solidity/helloworld.sol
```

You'll likely see the following output:

```
/solidity/helloworld.sol:6:5: Warning: No visibility specified. Defau
    function mortal() { owner = msg.sender; }
    ^---------------------------------------^
/solidity/helloworld.sol:9:5: Warning: No visibility specified. Defau
    function kill() { if (msg.sender == owner) selfdestruct(owner); }
    ^---------------------------------------------------------------^
/solidity/helloworld.sol:22:5: Warning: No visibility specified. Defa
    function greet() constant returns (string) {
    ^
Spanning multiple lines.
/solidity/helloworld.sol:1:1: Warning: Source file does not specify r
contract mortal {
^
Spanning multiple lines.
```

Since this is just an example smart contract, you can safely ignore these **warnings**.

After running `solc`, you should now see the following additional files in your directory:

```
greeter.abi
greeter.bin
greeter.signatures
mortal.abi
mortal.bin
mortal.signatures
```

We'll be using all 3 of the `greeter` files soon.

Congratulations -- you've just compiled a smart contract!

Step 4 - Deploy the smart contract

In this step, we'll be deploying the smart contract onto our 3-node **regtest** Qtum blockchain.

But first, a small administrative detour...

Using the node/wallet that has a positive `balance` (from Step 1), let's designate a specific address that will own the smart contract. We'll use the `getaccountaddress` RPC command to do this and call the account for this address `greeter_owner`:

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli
getaccountaddress greeter_owner
```

This should return a Quantum address (e.g. **qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**, but yours will surely be different) and if you run the same command again (with the same account name), you'll get back the same Quantum address.

You'll notice the Quantum address above begins with a lowercase `q` -- this distinguishes the address from a Quantum **mainnet** address (which starts with a capital `Q`).

You can view the accounts (and their balances) in your wallet by running:

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli listaccounts
```

You should see an entry with: `"greeter_owner": 0.00000000` -- which means the account `greeter_owner` has a balance of `0.00000000`.

Take note of the QTUM address that was returned above (e.g. **qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5** above, for me) -- we'll be using it often.

Ok, let's get on with deploying the contract...

The `greeter.bin` file that we generated in Step 3 contains the binary code (in hexadecimal) of the `greeter` contract that will be executed on Qtum's Ethereum VM.

You'll notice that the **greeter** contract has the following constructor:

```
function greeter(string _greeting)
```

Since the `greeter` contract constructor takes one (string) parameter, we'll want to provide this parameter to the smart contract when we deploy it. In order to do so, we'll need to get the binary representation of the string parameter we wish to give. This is where the `ethabi` tool is helpful. Run the following command to encode the "**Hello World!**" string:

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest encode
params -v string 'Hello World!' --lenient
```

It should output:

```
000000000000000000000000000000000000000000000000000000000000002000000000
000000000000000000000000000000000000000000000000000000000c48656c6c6f205
76f726c64210000000000000000000000000000000000000000
```

The above is essentially a binary (hexadecimal) representation of the `Hello World!` string. Feel free to provide your own greeting (NOTE: since this will be stored on the blockchain, larger strings will cost more to store).

Now, we'll need to concatenate the contents of `greeter.bin` with the output above -- on macOS/OSX and Linux, this can be done using:

```
$ cat greeter.bin && docker run --rm -v ${PWD}:/ethabi
cryptominder/ethabi:latest encode params -v string 'Hello World!' --
lenient
```

I won't reproduce the long string that the above outputs here, but it should start with:

```
6060604052341561000f57600080fd5b ...
```

... and end with the output got for the `Hello World!` parameter encoding.

To deploy the smart contract, we'll be using the `createcontract` RPC command. It takes the following arguments:

```
createcontract "bytecode" (gaslimit gasprice "senderaddress" broadcas
Create a contract with bytcode.

Arguments:
1. "bytecode"  (string, required) contract bytcode.
2. gasLimit  (numeric or string, optional) gasLimit, default: 2500000
3. gasPrice  (numeric or string, optional) gasPrice QTUM price per ga
4. "senderaddress" (string, optional) The quantum address that will b
5. "broadcast" (bool, optional, default=true) Whether to broadcast th
```

The `bytecode` is the long string (starting with `6060604052341561000f57600080fd5b`). For the `gasLimit`, we'll use the default of `2500000`. For the `gasPrice` we'll use `0.00000049`. And for `senderaddress`, we'll use the Quantum address (e.g. **qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**, for me). We won't set a value for `broadcast` since we'll be using the default value.

The command is therefore:

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli
createcontract
60606040523415610000f57600080fd5b6040516103173803806103178339810160405280
0805160008054600160a060020a0319163360 0160a060020a03161790555919091019050
6001818051610059929160200190610060565b50506100fb565b828054600181600116 1
5610100020316600290049060005260206000209060 1f016020900481019282601f1061
00a157805160ff1916838001178555610 0ce565b8280016001018555821561 00ce57918
2015b828111156100ce5782518255916020019190600101906100b3565b506100da9291
506100de565b5090565b6100f891905b80821115610 0da5760008155600101610 0e4565
b90565b61020d8061010a6000396000f30060606040526 3ffffffff7c010000000000000
00000000000000000000000000000000000000000000060003504166341c0e1b58114610
047578063cfae3217146100 5c57600080fd5b34156100525 7600080fd5b61005a6100e6
565b005b34156100675 7600080fd5b61006f610127565b60405160208082528190810 18
3818151815260200191508051906020019080838360005b838110156100ab5780820151
838201526020016100935 65b50505050905090810190601f168015610 0d857808203805
```

```
16001836020036101000a03191681526020019150 5b509250505060405180910390f35b
6000543373ffffffffffffffffffffffffffffffffffffffff908116911614156101255
760005473ffffffffffffffffffffffffffffffffffffffff16ff5b565b61012f6101cf
565b600180546001816001161561010002031660029004806 01f0160208091040260200
1604051908101604052809291908181526020018280546001816001161561010002 0316
600290048015610 1c55780601f1061019a5761010080835404028352916020019161 01c
5565b820191906000526020600020905b815481529060010190602001808311610 1a857
829003601f168201915b505050505090509056 5b6020604051908101604052600081529
05600a165627a7a723058209a62630a1678b0014fdfe901ed4f21cd251e9b7863cfccbf
79b1870bcc2e1de10029000000000000000000000000000000000000000000000000000
00000000000200000000000000000000000000000000000000000000000000000000000
00000c48656c6c6f20576f726c64210000000000000000000000000000000000000000
2500000 0.00000049 qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5
```

After running this command, we'll likely get en error saying: `Sender address does not have any unspent outputs`. This is because we have no UTXOs for this address. We'll fix this by sending 3 QTUM from our wallet to that specific address:

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli sendtoaddress
qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5 3
```

The command above will return a transaction id (e.g. `7d84a578a1d56ca71b1ea85b2028c34cce03bafe65dd782d3d8e41884eefa471`).

After a few moments (e.g. for at least 1 block to be generated), use the `listunspent` RPC command to verify that the address (e.g. **qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**, for me) has unspent transaction outputs (UTXOs):

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli listunspent 1
9999999 [\"qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5\"]
```

The above should output something like this:

```
[
  {
    "txid": "5b9b376deef5c10e31acca1f2ff0be64179878048de753e20c2a0490
    "vout": 0,
    "address": "qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5",
    "account": "greeter_owner",
    "scriptPubKey": "76a914002edb387c05038b700f97ce9dc40e305805c8df88
    "amount": 3.00000000,
    "confirmations": 1,
    "spendable": true,
    "solvable": true
  }
]
```

Notice the last parameter to `listunspent` (i.e. `[\"qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5\"]`) is a JSON array with the double-quotes (`"`) escaped with `\`.

Now that the address has an unspent output (i.e. an UTXO), we can re-run the `createcontract` command above. It should now output something like the following:

```
{
  "txid": "85d3c46886790cc164291500f3ed6bed20792c307b666a6fc490bd16c8
  "sender": "qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5",
  "hash160": "002edb387c05038b700f97ce9dc40e305805c8df",
  "address": "fd648ac3e7f89fd049507602d3d025cc90000606"
}
```

Congratulations again - you've just deployed a smart contract!

Step 5 - Inspecting the deployed smart contract

Referring back to the output of `createcontract` (in the previous step), you'll notice that the `sender` is the same as the Quantum address that we gave. The `hash160` value is simply a hash of the `sender`, and you can use the `fromhexaddress` and `gethexaddress` Quantum RPC commands to go from one form to the other.

The `address` field contains the address of the smart contract on the blockchain. If you run `listcontracts` again (like we did in step 1) -- you should see a new entry, e.g.:

```
"fd648ac3e7f89fd049507602d3d025cc90000606": 0.00000000,
```

The contract `address` is shown here too.

Now, let's use the `getaccountinfo` RPC command to retrieve information about our newly deployed smart contract (NOTE - use the `address` that you received from `createcontract` in Step 4, instead of **fd648ac3e7f89fd049507602d3d025cc90000606** that I'm using):

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli
getaccountinfo fd648ac3e7f89fd049507602d3d025cc90000606
```

The above will output something like:


```
{
  "address": "fd648ac3e7f89fd049507602d3d025cc90000606",
  "balance": 0,
  "storage": {
    "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563
      "0000000000000000000000000000000000000000000000000000000000000
    },
    "b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6
      "0000000000000000000000000000000000000000000000000000000000000
    }
  },
  "code": "606060405263ffffffff7c010000000000000000000000000000000000
}
```


From the above output, we can see that the smart contract has a `balance` of `0`, and that there are stored 2 items...

So, what is stored there? You'll notice that our smart contract had 2 variables (in `helloworld.sol`):

1. `owner` (inherited from **mortal**)
2. `greeting` (declared in **greeter**)

As you might guess, the first stored item (i.e. indexed by `0000000000000000000000000000000000000000000000000000000000000000`) stores the owner address. You'll recognize that it contains the `hash160` address of the sender. This means that our sender is the owner of this smart contract.

What about the second stored item (i.e. indexed by `0000000000000000000000000000000000000000000000000000000000000001`)? This is pretty much hexadecimal for `Hello World!`. You can use an online tool such as http://www.rapidtables.com/convert/number/hex-to-ascii.htm to convert it to ASCII, or run:

```
$ echo
48656c6c6f20576f726c642100000000000000000000000000000000000000018 | xxd
-r -p
```

to see that it will display `Hello World!`.

Step 6 - Invoking the smart contract

In this step, we'll be calling the `greet` function in our `greeter` smart contract.

To do this, we'll need to get the encoding of the `greet` function, and we can use the `ethabi` tool for this:

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest encode
function /ethabi/greeter.abi greet
```

The command above should return: `cfae3217`.

Alternatively, we can just use the `greeter.signatures` file that we generated when we used the `--hashes` option of `solc` in Step 3, e.g.

```
$ cat greeter.signatures
```

contains:

```
cfae3217: greet()
41c0e1b5: kill()
```

The `ethabi` tool would be useful if we wanted to encode parameters to a function. However, since this contract has no functions that takes a parameter (aside from the constructor), we can rely on the `greeter.signatures` file.

Ok, so the `greet` function has an signature of `cfae3217`.

Let's now use **qtumd_node2** to invoke the `greet` function of this smart contract. We'll invoke the `greet` function using the `callcontract` RPC command which takes the following parameters:

```
callcontract "address" "data" ( address )


Argument:
1. "address"          (string, required) The account address
2. "data"             (string, required) The data hex string
3. address              (string, optional) The sender address hex str
4. gasLimit             (string, optional) The gas limit for executin
```

For the `address` parameter, we'll use the address that was returned when we deployed the smart contract (e.g. **fd648ac3e7f89fd049507602d3d025cc90000606**, for me). For the `data`, it's simply the encoding for the `greet` function -- i.e. `cfae3217`. We can leave out the last 2 parameters.

Our command (using **qtumd_node2**) is therefore:

```
docker run -i --network container:qtumd_node2 -v
${PWD}/node2_qtumd.conf:/home/qtum/qtum.conf:ro -v
```

```
${PWD}/node2_data:/data cryptominder/qtum:latest qtum-cli callcontract
fd648ac3e7f89fd049507602d3d025cc90000606 cfae3217
```

The above will return something like the following:


```
{
  "address": "fd648ac3e7f89fd049507602d3d025cc90000606",
  "executionResult": {
    "gasUsed": 22737,
    "excepted": "None",
    "newAddress": "fd648ac3e7f89fd049507602d3d025cc90000606",
    "output": "00000000000000000000000000000000000000000000000000000
    "codeDeposit": 0,
    "gasRefunded": 0,
    "depositSize": 0,
    "gasForDeposit": 0
  },
  "transactionReceipt": {
    "stateRoot": "a5891d6126e68e032b640c71a75da231e1f0e159cd7dfc21c14
    "gasUsed": 22737,
    "bloom": "000000000000000000000000000000000000000000000000000000
    "log": [
    ]
  }
}
```


We're most interested in the `output` value. We'll use the `ethabi` tool to decode it, i.e.:

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest decode
params -t string
00000000000000000000000000000000000000000000000000000000000000200000000
0000000000000000000000000000000000000000000000000000000c48656c6c6f2057
6f726c6421000000000000000000000000000000000000000
```

will output:

```
string Hello World!
```

Bravo! You've now invoked a smart contract.

Step 7 - Clean-Up Time & callcontract vs. sendtocontract

The other function in the smart contract is `kill`. Before we invoke that function, I first wanted to talk about two similar commands in the **qtum-cli**: `callcontract` and `sendtocontract`.

callcontract vs. sendtocontract

The differences between these two commands are nicely documented elsewhere , but I'll repeat it here:

- `callcontract` - This will interact with an already deployed smart contract on the Qtum blockchain, with all computation taking place off-chain and no persistence to the blockchain. **This does not require gas.**
- `sendtocontract` - This will interact with an already deployed smart contract on the Qtum blockchain. All computation takes place on-chain and any state changes will be persisted to the blockchain. This allows tokens to be sent to a smart contract. **This requires gas.**

In the previous step, we used the `callcontract` RPC command to call `greet` since we didn't need to persist anything on the blockchain (i.e. we just retrieved a stored value). With the `kill` smart contract function, we'll be making changes to the state of the blockchain by removing the smart contract address. **NOTE:** The smart contract address will still exist on the blockchain (i.e. you can look it up with a block explorer), but it will no longer be indexed as an account.

Time to kill

As we just learnt, we must use the `sendtocontract` RPC command for the `kill` function. It has the following syntax:

```
sendtocontract "contractaddress" "data" (amount gaslimit gasprice sen
Send funds and data to a contract.

Arguments:
1. "contractaddress" (string, required) The contract address that wil
2. "datahex"  (string, required) data to send.
3. "amount"      (numeric or string, optional) The amount in QTUM to
4. gasLimit  (numeric or string, optional) gasLimit, default: 250000,
5. gasPrice  (numeric or string, optional) gasPrice Qtum price per ga
6. "senderaddress" (string, optional) The quantum address that will b
7. "broadcast" (bool, optional, default=true) Whether to broadcast th
```

For the `contractaddress`, we'll once again use the smart contract address (i.e. **fd648ac3e7f89fd049507602d3d025cc90000606**, for me). For the `data`, we'll use the hex code for the `kill` function -- which we saw earlier is `41c0e1b5` (e.g. in `greeter.signatures`).

Using **qtumd_node1**, our command is:

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli
sendtocontract fd648ac3e7f89fd049507602d3d025cc90000606 41c0e1b5
```

If we call the `getaccountinfo` RPC command on the contract address, we should find that it no longer exists....:

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli
getaccountinfo fd648ac3e7f89fd049507602d3d025cc90000606
```

… oh, oooops… it still exists!

What happened? If you look at the `kill` function inherited by the `greeter` contract, you'll see that it checks to ensure that the `sender` is also the `owner`. As such, the execution did nothing useful.

Before we simply set the `senderaddress`, you also need to **check that the sender address has at least 1 UTXO**, otherwise you'll get another `Sender address does not have any unspent outputs` message. It is important to know that we lost the UTXO that we created earlier when we did a `createcontract` call. Refer back to the `sendtoaddress` and `listunspent` RPC commands from Step 4 your to check your UTXOs and/or add another 3 QTUM to our address. In a future tutorial, I'll discuss how gas refunds are processed (which will also come back as UTXOs).

With the sender address now having at least 1 UTXO, we need to make sure to set the `senderaddress` in our `sendtocontract` call. Let's try that again using `amount` of `0` (since the function is not identified as **payable** in Solidity), a `gasLimit` of `250000`, a `gasPrice` of `0.00000049`, and setting the value of `senderaddress` appropriately (i.e. **qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**, for me):

```
$ docker run -i --network container:qtumd_node1 -v
${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v
${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli
sendtocontract fd648ac3e7f89fd049507602d3d025cc90000606 41c0e1b5 0
250000 0.00000049 qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5
```

Now if we call `getaccountinfo` again, we'll see that it returns an error saying: `Address does not exist.` Success!

This completes our coverage of Quantum's version of the **Ethereum Greeter Tutorial** as found at https://ethereum.org/greeter , simply using `qtum-cli`, `solc`, and `ethabi`.

References / Further Reading

For much of the content in this tutorial, I've relied on documentation provided by Jordan Earls , the lead developer for Quantum. In particular, I found the following helpful:

- The Qtum Sparknet Faucet
- Qtum Sparknet Usage and Information

What's Next?

In upcoming tutorials, I'll be discussing many other topics, including:

- Ethereum VM Logging.
- Data types (e.g. mapping, struct, etc.)
- Payable contracts and functions.
- Taking a closer look at selfdestruct.
- Gas estimation and gas refunds.
- Using an SPV client.
- Creating a token.
- Calling another contract.
- Global variables (e.g. msg, tx, block)
- ... and more.

In the meantime, you can also expect to hear about advancements from the Quantum team.

If you need help, please use the comment section below, or look for me on the Qtum subreddit . I'm also active on Quantum's Slack (although Slack invitations are closed as the Quantum team prepares to move to another collaboration/messaging platform).

Thank You!

If you found this tutorial useful, and you are so inclined, my Quantum tip wallet address (running on my Raspberry Pi 3 (https://steemit.com/qtum/@cryptominder/qtum-staking-tutorial-using-qtumd-on-a-raspberry-pi-3)) is: **QUa3yA8ALfQyM5eEb9sDPRWkX6sSurMs6D** .

I appreciate your support!

#blockchain (/trending/blockchain) #smartcontract (/trending/smartcontract) #ethereum (/trending/ethereum) #docker (/trending/docker)

3 years ago in #qtum (/trending/qtum) by Reply 7

cryptominder (38) (/@cryptominder) (/qtum/@cryptominder/quantum-qtum-blockchain-developer-tutorial-hello-world)

$1.48 13 votes

Sort: Trending

moxi (42) (/@moxi) 2 years ago (/qtum/@moxi/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180113t081410351z#@moxi/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180113t081410351z) [-]

Brilliant! Awesome tutorial. We miss you cryptominder! Please come back ;)

$0.00 Reply

moxi (42) (/@moxi) 2 years ago (/qtum/@moxi/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180228t045708908z#@moxi/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180228t045708908z) (edited) [-]

In Step 4 after creating the smart contract we do not get a message saying "Sender address does not have any unspent outputs." Instead the contract is deployed but under a completely different sender address and thus hash address. From here I'm not able to ever delete this contract even when using the sender address in the kill operation.

I'm banging my head on this trying to understand it. We need you cryptominder, you're our only hope. ;)

$0.00 Reply

moxi (42) (/@moxi) 2 years ago (/qtum/@moxi/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180228t045708908z#@moxi/re-moxi-re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180228t055543212z) [-]

I waited a while and for some reason the contract is now killed when I use the newly generated sender address. However, each time I create a contract I still get a brand new address that is not greeter_owner even though I've passed the greeter_owner address to createcontract. Not sure why it's not deployed under greeter_owner.

$0.00 Reply

dragonplan (30) (/@dragonplan) 2 years ago (/qtum/@dragonplan/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180719t092451823z#@dragonplan/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20180719t092451823z) [-]

Hi , am from China, your article is very good.
i want to translate it and share your idea with people around me.
is that possible ?
Thank you so much for your sharing.

$0.00 Reply

steemitboard (66) (/@steemitboard) 2 years ago (/qtum/@steemitboard/steemitboard-notify-cryptominder-20180721t185447000z#@steemitboard/steemitboard-notify-cryptominder-20180721t185447000z) [-]

Congratulations @cryptominder (/@cryptominder)! You have received a personal award!

1 Year on Steemit

Click on the badge to view your Board of Honor.

Do not miss the last post from @steemitboard (/@steemitboard):
SteemitBoard World Cup Contest - The results, the winners and the prizes (https://steemit.com/steemitboard/@steemitboard/steemitboard-world-cup-contest-the-results-and-prizes)

> Do you like SteemitBoard's project (https://steemit.com/@steemitboard)? Then **Vote for its witness** and **get one more award**!

$0.00 Reply

gauravagrwal (25) (/@gauravagrwal) 2 years ago (/qtum/@gauravagrwal/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20181004t114458597z#@gauravagrwal/re-cryptominder-quantum-qtum-blockchain-developer-tutorial-hello-world-20181004t114458597z) [-]

Hey Awesome tutorials, can we republish these tutorials on Coinmonks?

https://medium.com/coinmonks

$0.00 Reply

steemitboard (66) (/@steemitboard) 10 months ago (/qtum/@steemitboard/steemitboard-notify-cryptominder-20190721t174957000z#@steemitboard/steemitboard-notify-cryptominder-20190721t174957000z) [-]

Congratulations @cryptominder (/@cryptominder)! You received a personal award!

Happy Birthday! - You are on the Steem blockchain for 2 years!

You can view your badges on your Steem Board and compare to others on the Steem Ranking

Vote for @Steemitboard as a witness to get one more award and increased upvotes!

$0.00 Reply