Super-Short Beta Docs (For Jthereum!)

Welcome

Welcome early beta tester, and Thank You!

The full Docs aren't ready yet, but let's get you started anyway.

Getting Started

Initial Setup

Here's what you need to set up first:

- A working JDK. Preferably version 8, but other versions *may* work
- A working binary executable Solidity compiler ('solc') in your path
- Preferably a Java IDE you are comfortable with (Such as IntelliJ)

Hopefully you're already setup with an IDE and a SDK. For the 'solc' binary you may be able to download one from here:

https://github.com/ethereum/solidity/releases

Unfortunately, they don't have a pre-built macOS binary. If you're on macOS you can build it with the commands:

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

Build instructions for all platforms can be found here: https://solidity.readthedocs.io/en/develop/installing-solidity.html#building-from-source .

If you have trouble with any of this, let us know and we'll help you out.

By the way, you can always get in touch with us quickly by going to our website (https://jthereum.com), and clicking on the little blue 'Intercom' button in the

lower right.

Installing Jthereum

First, you need a beta release of the jthereum.jar file. If you don't have that already, go to https://jthereum.com and click on the blue Intercom button on the lower right and we'll get you hooked up.

To install, run the command:

```
java -jar jthereum.jar
```

This will set up a directory with the files you will need, and it will also create initial Ethereum accounts for you on the Ropsten test network, and also on our test network. It will also pre-fund those accounts with some ETH for you. Neat!

On macOS the Jthereum directory will be at ~/Library/Jthereum. On all other systems it will be at ~/Jthereum.

Take a look in that directory, and you'll see something like this:

```
% ls -l
drwxr-xr-x  3 nat  staff        96 Jul 12 23:27 docs
-rw-r--r--  1 nat  staff      1225 Jul 12 23:27 jthereum.config
-rw-r--r--  1 nat  staff  24535242 Jul 12 23:27 jthereum.jar
drwxr-xr-x  3 nat  staff        96 Jul 12 23:27 providedSource
```

Yep, the jar file copied itself to this location. You can delete the original copy if you want.

At this point you're ready to build and deploy your first contract!

Building a simple project

Ok, hopefully you already know Java... if not then maybe this isn't a good idea. Go learn Java first, then come back. I'll wait.

Great! That was quick. Now, use your IDE to create a sample project, and add the jthereum.jar file to your project as a library.

You can put a copy of jthereum.jar into your project directory if you want, or you can just reference the copy that is in the Jthereum installation directory.

You can always find a copy of your jthereum.jar file in the installation directory (~/Library/Jthereum on macOS, ~/Jthereum on everything else).

Ok, you got your project set up and ready to go... Now create your first Smart Contract class:

```
package com.u7.jthereum.exampleContracts;

import com.u7.jthereum.annotations.*;
import static com.u7.jthereum.Jthereum.*;

public class SimpleSetValueContract
{
    int value;

    public void setValue(final int newValue)
    {
        value = newValue;
    }

    @View
    public int getValue()
    {
        return value;
    }

    public static void main(final String[] args)
    {
        compileAndDeploy();

        final SimpleSetValueContract a =
(SimpleSetValueContract)createProxy();

        a.setValue(7);

        final int value = a.getValue();

        p("Got value: " + value);
    }
}
```

Now run it!

I'm going to assume everything worked ok, if not get in touch with our helpful tech support staff.

Ok, so what just happened?

Let's break it down:

- When you called compileAndDeploy():
  - Jthereum took your class and *translated* it into Solidity
  - That Solidity code was then compiled with a Solidity compiler
  - The resulting EVM bytecodes were then deployed to the blockchain
- When you called createProxy():
  - A special proxy object was created that is derived from your class
  - The proxy is linked to the contract you just deployed.
  - Calls to the proxy object are translated into blockchain transactions
- When you called a.setValue(7):
  - This call was translated into a blockchain transaction and executed
  - This resulted in the value '7' being stored in the 'value' storage location on the blockchain
- When you called a.getValue():
  - This queried the state of the blockchain to discover the current contents of 'value'

Here is the expected output:

```
Compiling com.u7.jthereum.exampleContracts.SimpleSetValueContract

Solidity Source:

// Generated by Jthereum BETA version!
pragma solidity ^0.5.2;
contract SimpleSetValueContract
{
    int32 value;
    function setValue(int32 newValue) public
    {
        value = newValue;
    }
    function getValue() public view returns (int32)
    {
        return value;
    }

}
```

```
Running command: solc --optimize --overwrite --bin --abi --hashes --
metadata -o ./soltmp ./soltmp/SimpleSetValueContract.sol
Compiler run successful. Artifact(s) can be found in directory ./soltmp.

Deploying contract to the blockchain

Tx hash:
0x08488cffdb395dce3327f416b0b852fb105ca79982ecfd779da4b8505a22c354
Waiting for transaction to be mined.
Your contract was deployed in block #2,500,764
The new contract address is at:
0x86b58f589e552d0b2cc4f13ff48d5d58ae9e6607

Calling setValue:

Tx hash:
0x49ae9c36e58736599a38b6e5d28ffa5a89931d2c82f429098ce39de82efae0a3
Waiting for transaction to be mined.
Gas Used: 41,897
Total WEI used: 41,897
Total ETH used: 0.000000
(Gas Price: 1 wei)
Transaction succeeded with status: 0x1

Got value: 7
```

Note that blockchain transactions are inherently asynchronous. Jthereum wraps those transactions inside a single call, so they appear to be synchronous. We'll shortly be adding a feature for more advanced users who wish operate with ansychronous transactions directly.

You may have a couple more questions about how this all works, so here are some answers:

Q: This seems too easy... How did the code know which class I wanted to compile? What account am I using? Where did the ETH come from?

A: Thanks, we tried to make things as easy as possible! Jthereum has a system which automatically gathers together the arguments that are needed for each operation. There are multiple overloads of each method call which take different numbers of arguments. If you leave some arguments out, Jthereum will do it's best to figure out values that make sense. In the best case you can call some methods with *no* arguments, and Jthereum will figure it all out.

For example, when you called compileAndDeploy(), Jthereum assumed that the class you wanted to compile was the one that contains the main() method. If your main() was in another class you would need to explicitly supply the Class object that you want Jthereum to compile.

When you called createProxy(), Jthereum knew the class in the same way as compileAndDeploy(). From that class it found the address for the most recently deployed instance of that class and used that.

If you want to know in detail where each argument is coming from, you can set a property in your 'jtherem.config' file (located in the Jthereum installation directory):

```
DUMP_ARGUMENT_DETAILS_TO_CONSOLE=true
```

There are a bunch of arguments that Jthereum attempts to figure out for you. Here's the list of arguments:

```
Class       contractClass;
String      blockchainName;
String      address;
BigInteger  privateKey;
String      web3jURL;
BigInteger  gasLimit;
BigInteger  gasPrice;
Boolean     verifySource;
```

You'll find the detailed description of how all this works in the full manual, but for now you don't have to worry about it. If you need to override any of these values you can usually do it via an explicit argument to one of the method variants, or by setting a property in either 'jthereum.config', or 'jthereum.properties'.

Hey, since we're talking about properties...

Property Files

The Jthereum installation directory (~/Library/Jthereum on macOS, or just ~/Jthereum on other platforms) contains a file 'jthereum.config'.

'jthereum.config' is a *global* property file. These values are used for every Jthereum based project on your system.

If you want to have different properties for each project, no problem. Simply copy the 'jtherem.config' file to a file named 'jthereum.properties' in your project directory (Note that this must also be the current directory in the context of your running application! Jthereum looks for ./jthereum.properties in the local directory first, and if doesn't find that then it falls back to {Jthereum Installation

Directory}/jthereum.config.

Here's a brief review of some of the properties.

```
sourceRootDirectory0 = ./src
sourceRootDirectory1 = ./ProvidedSource

#DUMP_ARGUMENT_DETAILS_TO_CONSOLE = true

DEFAULT_BLOCKCHAIN = ropsten

ROPSTEN_PrivateKey = {some long hex number}
ROPSTEN_Web3j_URL =
https://ropsten.infura.io/v3/876ec9dc7dc64fbca7524e8b1066267c
ROPSTEN_VERIFY_SOURCE = true


TEST_PrivateKey = {some long hex number}
TEST_Web3j_URL = http://test.jthereum.com:8545


# MAINNET   LIVE!!!!  Please be cautious with connecting to mainnet as
# transactions will consume real ETH!
MAINNET_PrivateKey = {Mainnet private key goes here}
MAINNET_Web3j_URL =
https://mainnet.infura.io/v3/876ec9dc7dc64fbca7524e8b1066267c
MAINNET_VERIFY_SOURCE = true
```

The sourceRootDirectory* properties tell Jthereum where to find the Java source code for your classes. You can have any number of these properties. The directories will be searched in the order you specify.

If you want to see the detailed arguments that Jthereum is using for each call, uncomment

```
DUMP_ARGUMENT_DETAILS_TO_CONSOLE = true
```

To tell Jthereum which blockchain to communicate with, set this property:

```
DEFAULT_BLOCKCHAIN = ropsten
```

Common blockchain names include:

```
test
```

```
ropsten
mainnet
```

You can also create your own blockchain names and set up properties for them. This way you can set up Jthereum to talk to your own private node/blockchain!

The blockchain specific properties include:

```
{NAME}_PrivateKey = {some long hex number}
{NAME}_Web3j_URL = https://{URL of node you want to connect to}
{NAME}_VERIFY_SOURCE = {true if you want to verify source on
ethercan.io}
```

These specify the default private key to use for transactions, the node to connect to, and if Jthereum should attempt to verify your contract source code on https://etherscan.io.

Where did the ETH come from?

When you installed Jthereum it created a couple test addresses for you and pre-funded those accounts with some ETH! That way you can get started without having to worry about how to get ETH.

Your 'ropsten' account was funded with 0.5 ETH, and your 'test' blockchain account was funded with 100,000,000 ETH.

When you're ready to do some work on 'mainnet' you'll need to get some *real* ETH. Basically you have to buy it. For now you can get used to Jthereum on our private test blockchain ('test'), and on the public test blockchain ('ropsten'). One cool thing about 'ropsten' is that you can see all the transactions up on https://ropsten.etherscan.io.

You can check your account balances with code like this:

```
package com.u7.internalTools;

import static com.u7.jthereum.Jthereum.*;

public class CheckJthereumBalances
{
    public static void main(final String[] args)
    {
        p(weiAmountToString(getBalance("test")));

        p(weiAmountToString(getBalance("ropsten")));
    }
```

```
}
```

Sample Code

In the Jthereum installation directory (~/Library/Jthereum on macOS, or just ~/Jthereum on other platforms) you'll see a sub-directory 'providedSource'. This directory contains source code from some sample contracts that you can compile and play around with. Feel free to copy this code into your projects and modify it as you see fit.

Thanks

Well, that's it for now. Thanks again for being a Beta Tester.

Let us know if you have any questions or suggestions on the product or the documentation.

We're very excited about Jthereum and we hope you like it!