web3j Documentation

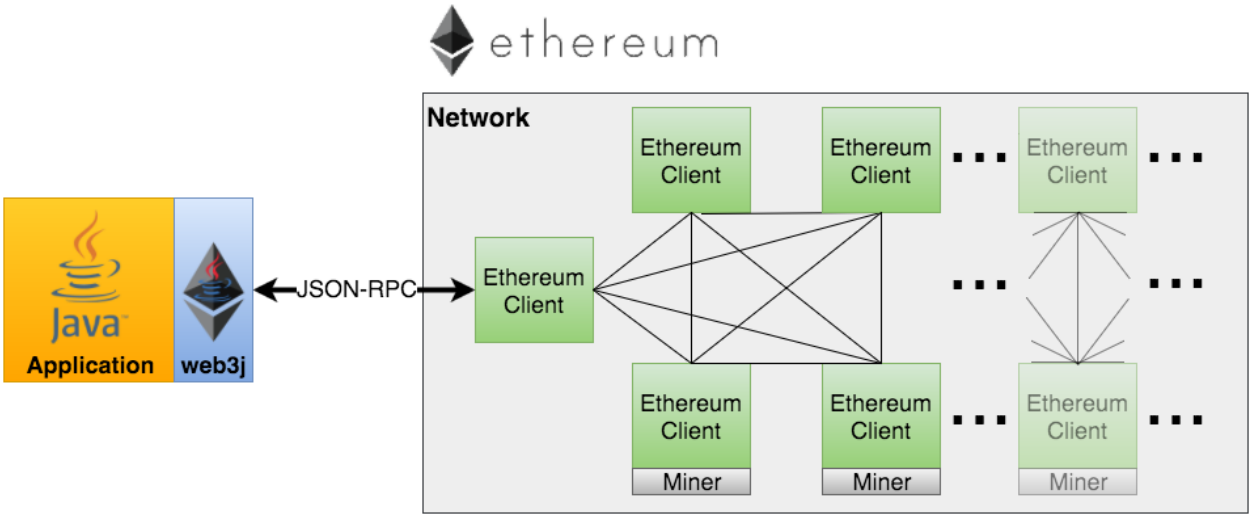
Release 2.3.1

Conor Svensson

September 06, 2017

Contents

web3j is a lightweight, reactive, type safe Java and Android library for integrating with clients (nodes) on the Ethereum network:



This allows you to work with the Ethereum blockchain, without the additional overhead of having to write your own integration code for the platform.

The Java and the Blockchain talk provides an overview of blockchain, Ethereum and web3j.

CHAPTER 1

Features

- Complete implementation of Ethereum's JSON-RPC client API over HTTP and IPC
- Ethereum wallet support
- Reactive-functional API for working with filters
- Auto-generation of Java smart contract wrappers to create, deploy, transact with and call smart contracts from native Java code
- Support for Parity's Personal, and Geth's Personal client APIs
- Support for Infura, so you don't have to run an Ethereum client yourself
- Comprehensive integration tests demonstrating a number of the above scenarios
- Command line tools
- Android compatible
- Support for JP Morgan's Quorum via web3j-quorum

CHAPTER 2

Dependencies

It has seven runtime dependencies:

- RxJava for its reactive-functional API
- Apache HTTP Client
- Jackson Core for fast JSON serialisation/deserialisation
- Bouncy Castle and Java Scrypt for crypto
- JavaPoet for generating smart contract wrappers
- Jnr-unixsocket for *nix IPC

CHAPTER 3

Donate

You can help fund the development of web3j by donating to the following wallet addresses:

Ethereum	0x2dfBf35bb7c3c0A466A6C48BEBf3eF7576d3C420
Bitcoin	1DfUeRWUy4VjekPmmZUNqCjcJBMwsyp61G

CHAPTER 4

Commercial support and training

Commercial support and training is available from blk.io.

CHAPTER 5

Contents:

Getting Started

Add the latest web3j version to your project build configuration.

Maven

Java 8:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>2.3.1</version>
</dependency>
```

Android:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core-android</artifactId>
  <version>2.2.1</version>
</dependency>
```

Gradle

Java 8:

```
compile ('org.web3j:core:2.3.1')
```

Android:

```
compile ('org.web3j:core-android:2.2.1')
```

Start a client

Start up an Ethereum client if you don't already have one running, such as Geth:

```
$ geth --fast --cache=512 --rpcapi personal,db,eth,net,web3 --rpc --testnet
```

Or Parity:

```
$ parity --chain testnet
```

Or use Infura, which provides **free clients** running in the cloud:

```
Web3j web3 = Web3j.build(new InfuraHttpService("https://morden.infura.io/your-token"));
```

For further information refer to Using Infura with web3j.

Start sending requests

To send asynchronous requests using a Future:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
Web3ClientVersion web3ClientVersion = web3.web3ClientVersion().sendAsync().get();
String clientVersion = web3ClientVersion.getWeb3ClientVersion();
```

To use an RxJava Observable:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
web3.web3ClientVersion().observable().subscribe(x -> {
    String clientVersion = x.getWeb3ClientVersion();
    ...
});
```

To send synchronous requests:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
Web3ClientVersion web3ClientVersion = web3.web3ClientVersion().send();
String clientVersion = web3ClientVersion.getWeb3ClientVersion();
```

Note: for Android use:

> Web3j web3 = Web3jFactory.build(new HttpService()); // defaults to http://localhost:8545/ ...

IPC

web3j also supports fast inter-process communication (IPC) via file sockets to clients running on the same host as web3j. To connect simply use the relevent *IpcService* implemntation instead of *HttpService* when you create your service:

```
// OS X/Linux/Unix:
Web3j web3 = Web3j.build(new UnixIpcService("/path/to/socketfile"));
...

// Windows
Web3j web3 = Web3j.build(new WindowsIpcService("/path/to/namedpipefile"));
...
```

Note: IPC is not currently available on web3j-android.

Filters

web3j functional-reactive nature makes it really simple to setup observers that notify subscribers of events taking place on the blockchain.

To receive all new blocks as they are added to the blockchain:

```
Subscription subscription = web3j.blockObservable(false).subscribe(block -> {
    ...
});
```

To receive all new transactions as they are added to the blockchain:

```
Subscription subscription = web3j.transactionObservable().subscribe(tx -> {
    ...
});
```

To receive all pending transactions as they are submitted to the network (i.e. before they have been grouped into a block together):

```
Subscription subscription = web3j.pendingTransactionObservable().subscribe(tx -> {
    ...
});
```

Or, if you'd rather replay all blocks to the most current, and be notified of new subsequent blocks being created:

```
Subscription subscription = catchUpToLatestAndSubscribeToNewBlocksObservable(
        <startBlockNumber>, <fullTxObjects>)
        .subscribe(block -> {
            ...
});
```

There are a number of other transaction and block replay Observables described in Filters and Events.

Topic filters are also supported:

```
EthFilter filter = new EthFilter(DefaultBlockParameterName.EARLIEST,
        DefaultBlockParameterName.LATEST, <contract-address>)
             .addSingleTopic(...)|.addOptionalTopics(..., ...)|...;
web3j.ethLogObservable(filter).subscribe(log -> {
    ...
});
```

Subscriptions should always be cancelled when no longer required:

```
subscription.unsubscribe();
```

Note: filters are not supported on Infura.

For further information refer to Filters and Events and the Web3jRx interface.

Transactions

web3j provides support for both working with Ethereum wallet files (recommended) and Ethereum client admin commands for sending transactions.

To send Ether to another party using your Ethereum wallet file:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
Credentials credentials = WalletUtils.loadCredentials("password", "/path/to/walletfile");
TransactionReceipt transactionReceipt = Transfer.sendFunds(
        web3, credentials, "0x...", BigDecimal.valueOf(1.0), Convert.Unit.ETHER);
```

Or if you wish to create your own custom transaction:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
Credentials credentials = WalletUtils.loadCredentials("password", "/path/to/walletfile");

// get the next available nonce
EthGetTransactionCount ethGetTransactionCount = web3j.ethGetTransactionCount(
            address, DefaultBlockParameterName.LATEST).sendAsync().get();
BigInteger nonce = ethGetTransactionCount.getTransactionCount();

// create our transaction
RawTransaction rawTransaction  = RawTransaction.createEtherTransaction(
            nonce, <gas price>, <gas limit>, <toAddress>, <value>);

// sign & send our transaction
byte[] signedMessage = TransactionEncoder.signMessage(rawTransaction, credentials);
String hexValue = Numeric.toHexString(signedMessage);
EthSendTransaction ethSendTransaction = web3j.ethSendRawTransaction(hexValue).sendAsync().get();
// ...
```

Although it's far simpler using web3j's *smart contract wrappers*.

Using an Ethereum client's admin commands (make sure you have your wallet in the client's keystore):

```
Parity parity = Parity.build(new HttpService());  // defaults to http://localhost:8545/
PersonalUnlockAccount personalUnlockAccount = parity.personalUnlockAccount("0x000...", "a password")
if (personalUnlockAccount.accountUnlocked()) {
    // send a transaction, or use parity.personalSignAndSendTransaction() to do it all in one
}
```

Working with smart contracts with Java smart contract wrappers

web3j can auto-generate smart contract wrapper code to deploy and interact with smart contracts without leaving Java.

To generate the wrapper code, compile your smart contract:

```
$ solc <contract>.sol --bin --abi --optimize -o <output-dir>/
```

Then generate the wrapper code using web3j's Command Line Tools:

```
web3j solidity generate /path/to/<smart-contract>.bin /path/to/<smart-contract>.abi -o /path/to/src/r
```

Now you can create and deploy your smart contract:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
Credentials credentials = WalletUtils.loadCredentials("password", "/path/to/walletfile");

YourSmartContract contract = YourSmartContract.deploy(
        <web3j>, <credentials>,
        GAS_PRICE, GAS_LIMIT,
        <initialEtherValue>,
        <param1>, ..., <paramN>).get();  // constructor params
```

Or use an existing:

```
YourSmartContract contract = YourSmartContract.load(
        "0x<address>", <web3j>, <credentials>, GAS_PRICE, GAS_LIMIT);
```

To transact with a smart contract:

```
TransactionReceipt transactionReceipt = contract.someMethod(
            new Type(...),
            ...).get();
```

To call a smart contract:

```
Type result = contract.someMethod(new Type(...), ...).get();
```

For more information refer to *Solidity smart contract wrappers*.

Further details

In the Java 8 build:

- web3j provides type safe access to all responses. Optional or null responses are wrapped in Java 8's Optional type.
- Async requests are handled using Java 8's CompletableFutures.

In both the Java 8 and Andriod builds:

- Quantity payload types are returned as BigIntegers. For simple results, you can obtain the quantity as a String via Response.getResult().

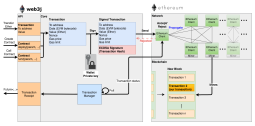
Transactions

Broadly speaking there are three types transactions supported on Ethereum:

1. *Transfer of Ether from one party to another*
2. *Creation of a smart contract*
3. *Transacting with a smart contract*

To undertake any of these transactions, it is necessary to have Ether (the fuel of the Ethereum blockchain) residing in the Ethereum account which the transactions are taking place from. This is to pay for the *Gas* costs, which is the transaction execution cost for the Ethereum client that performs the transaction on your behalf, comitting the result to the Ethereum blockchain. Instructions for obtaining Ether are described below in *Obtaining Ether*.

Additionally, it is possible to query the state of a smart contract, this is described in *Querying the state of a smart contract*.



Obtaining Ether

To obtain Ether you have two options:

1. Mine it yourself
2. Buy Ether from another party

Mining it yourself in a private environment, or the public tet environment (testnet) is very straight forwards. However, in the main live environment (mainnet) it requires significant dedicated GPU time which is not likely to be feasible unless you already have a gaming PC with multiple dedicated GPUs. If you wish to use a private environment, there is some guidance on the Homestead documentation.

To purchase Ether you will need to go via an exchange. As different regions have different exchanges, you will need to research the best location for this yourself. The Homestead documentation contains a number of exchanges which is a good place to start.

Alternatively, if you need some Ether on testnet to get started, please post a message with your wallet address to the web3j Gitter channel and I'll send you some.

Mining on testnet/private blockchains

In the Ethereum test environment (testnet), the mining difficulty is set lower then the main environment (mainnet). This means that you can mine new Ether with a regular CPU, such as your laptop. What you'll need to do is run an Ethereum client such as Geth or Parity to start building up reserves. Further instructions are available on the respective sites.

Geth https://github.com/ethereum/go-ethereum/wiki/Mining

Parity https://github.com/paritytech/parity/wiki/Mining

Once you have mined some Ether, you can start transacting with the blockchain.

Gas

When a transaction takes place in Ethereum, a transaction cost must be paid to the client that executes the transaction on your behalf, committing the output of this transaction to the Ethereum blockchain.

This cost is measure in gas, where gas is the number of instructions used to execute a transaction in the Ethereum Virtual Machine. Please refer to the Homestead documentation for further information.

What this means for you when working with Ethereum clients is that there are two parameters which are used to dictate how much Ether you wish to spend in order for a tranaction to complete:

Gas price

> This is the amount you are prepared in Ether per unit of gas. It defaults to a price of 9000 Wei (9×10^{-15} Ether).

Gas limit

> This is the total amount of gas you are happy to spend on the transaction execution. There is an upper limit of how large a single transaction can be in an Ethereum block which restricts this value typically to less then 1,500,000. The current gas limit is visible at https://ethstats.net/.

These parameters taken together dictate the maximum amount of Ether you are willing to spend on transaction costs. i.e. you can spend no more then *gas price * gas limit*. The gas price can also affect how quickly a transaction takes place depending on what other transactions are available with a more profitable gas price for miners.

You may need to adjust these parameters to ensure that transactions take place in a timely manner.

Transaction mechanisms

When you have a valid account created with some Ether, there are two mechanisms you can use to transact with Ethereum.

1. *Transaction signing via an Ethereum client*
2. *Offline transaction signing*

Both mechanisms are supported via web3j.

Transaction signing via an Ethereum client

In order to transact via an Ethereum client, you first need to ensure that the client you're transacting with knows about your wallet address. You are best off running your own Ethereum client such as Geth/Parity in order to do this. Once you have a client running, you can create a wallet via:

- The Geth Wiki contains a good run down of the different mechanisms Geth supports such as importing private key files, and creating a new account via it's console
- Alternatively you can use a JSON-RPC admin command for your client, such as personal_newAccount for Parity or Geth

With your wallet file created, you can unlock your account via web3j by first of all creating an instance of web3j that supports Parity/Geth admin commands:

```
Parity parity = Parity.build(new HttpService());
```

Then you can unlock the account, and providing this was successful, send a transaction:

```
PersonalUnlockAccount personalUnlockAccount = parity.personalUnlockAccount("0x000...", "a password")
if (personalUnlockAccount.accountUnlocked()) {
    // send a transaction
}
```

Transactions for sending in this manner should be created via EthSendTransaction, with the Transaction type:

```
Transaction transaction = Transaction.createContractTransaction(
                <from address>,
                <nonce>,
                BigInteger.valueOf(<gas price>),
                "0x...<smart contract code to execute>"
        );

        org.web3j.protocol.core.methods.response.EthSendTransaction
                transactionResponse = parity.ethSendTransaction(ethSendTransaction)
                .sendAsync().get();

        String transactionHash = transactionResponse.getTransactionHash();

        // poll for transaction response via org.web3j.protocol.Web3j.ethGetTransactionReceipt(<txHash
```

Where the <*nonce*> value is obtained as per *below*.

Please refer to the integration test DeployContractIT and its superclass Scenario for further details of this transaction workflow.

Further details of working with the different admin commands supported by web3j are available in the section Management APIs.

Offline transaction signing

If you'd prefer not to manage your own Ethereum client, or do not want to provide wallet details such as your password to an Ethereum client, then offline transaction signing is the way to go.

Offline transaction signing allows you to sign a transaction using your Ethereum Ethereum wallet within web3j, allowing you to have complete control over your private credentials. A transaction created offline can then be sent to any Ethereum client on the network, which will propagate the transaction out to other nodes, provided it is a valid transaction.

Creating and working with wallet files

In order to sign transactions offline, you need to have either your Ethereum wallet file or the public and private keys associated with an Ethereum wallet/account.

web3j is able to both generate a new secure Ethereum wallet file for you, or work with an existing wallet file.

To create a new wallet file:

```
String fileName = WalletUtils.generateNewWalletFile(
        "your password",
        new File("/path/to/destination"));
```

To load the credentials from a wallet file:

```
Credentials credentials = WalletUtils.loadCredentials(
        "your password",
        "/path/to/walletfile");
```

These credentials are then used to sign transactions.

Please refer to the Web3 Secret Storage Definition for the full wallet file specification.

Signing transactions

Transactions to be used in an offline signing capacity, should use the RawTransaction type for this purpose. The RawTransaction is similar to the previously mentioned Transaction type, however it does not require a *from* address, as this can be inferred from the signature.

In order to create and sign a raw transaction, the sequence of events is as follows:

1. Identify the next available *nonce* for the sender account
2. Create the RawTransaction object
3. Encode the RawTransaction object
4. Sign the RawTransaction object
5. Send the RawTransaction object to a node for processing

The nonce is an increasing numeric value which is used to uniquely identify transactions. A nonce can only be used once and until a transaction is mined, it is possible to send multiple versions of a transaction with the same nonce, however, once mined, any subsequent submissions will be rejected.

Once you have obtained the next available *nonce*, the value can then be used to create your transaction object:

```
RawTransaction rawTransaction  = RawTransaction.createEtherTransaction(
             nonce, <gas price>, <gas limit>, <toAddress>, <value>);
```

The transaction can then be signed and encoded:

```
byte[] signedMessage = TransactionEncoder.signMessage(rawTransaction, <credentials>);
String hexValue = Numeric.toHexString(signedMessage);
```

Where the credentials are those loaded as per *Creating and working with wallet files*.

The transaction is then sent using eth_sendRawTransaction:

```
EthSendTransaction ethSendTransaction = web3j.ethSendRawTransaction(hexValue).sendAsync().get();
String transactionHash = ethSendTransaction.getTransactionHash();
// poll for transaction response via org.web3j.protocol.Web3j.ethGetTransactionReceipt(<txHash>)
```

Please refer to the integration test CreateRawTransactionIT for a full example of creating and sending a raw transaction.

The transaction nonce

The nonce is an increasing numeric value which is used to uniquely identify transactions. A nonce can only be used once and until a transaction is mined, it is possible to send multiple versions of a transaction with the same nonce, however, once mined, any subsequent submissions will be rejected.

You can obtain the next available nonce via the eth_getTransactionCount method:

```
EthGetTransactionCount ethGetTransactionCount = web3j.ethGetTransactionCount(
             address, DefaultBlockParameterName.LATEST).sendAsync().get();

    BigInteger nonce = ethGetTransactionCount.getTransactionCount();
```

The nonce can then be used to create your transaction object:

```
RawTransaction rawTransaction  = RawTransaction.createEtherTransaction(
             nonce, <gas price>, <gas limit>, <toAddress>, <value>);
```

Transaction types

The different types of transaction in web3j work with both Transaction and RawTransaction objects. The key difference is that Transaction objects must always have a from address, so that the Ethereum client which processes the eth_sendTransaction request know which wallet to use in order to sign and send the transaction on the message senders behalf. As mentioned *above*, this is not necessary for raw transactions which are signed offline.

The subsequent sections outline the key transaction attributes required for the different transaction types. The following attributes remain constant for all:

- Gas price
- Gas limit
- Nonce
- From

Transaction and RawTransaction objects are used interchangeably in all of the subsequent examples.

Transfer of Ether from one party to another

The sending of Ether between two parties requires a minimal number of details of the transaction object:

to the destination wallet address

value the amount of Ether you wish to send to the destination address

```
BigInteger value = Convert.toWei("1.0", Convert.Unit.ETHER).toBigInteger();
RawTransaction rawTransaction  = RawTransaction.createEtherTransaction(
             <nonce>, <gas price>, <gas limit>, <toAddress>, value);
// send...
```

Creation of a smart contract

To deploy a new smart contract, the following attributes will need to be provided

value the amount of Ether you wish to deposit in the smart contract (assumes zero if not provided)

data the hex formatted, compiled smart contract creation code

```
// using a raw transaction
RawTransaction rawTransaction = RawTransaction.createContractTransaction(
        <nonce>,
        <gasPrice>,
        <gasLimit>,
        <value>,
        "0x <compiled smart contract code>");
// send...

// get contract address
EthGetTransactionReceipt.TransactionReceipt transactionReceipt = sendTransactionReceiptRequest(transa

Optional<String> contractAddressOptional = transactionReceipt.getContractAddress();
```

If the smart contract contains a constructor, the associated constructor field values must be encoded and appended to the *compiled smart contract code*:

```
String encodedConstructor =
             FunctionEncoder.encodeConstructor(Arrays.asList(new Type(value), ...));

// using a regular transaction
Transaction transaction = Transaction.createContractTransaction(
        <fromAddress>,
        <nonce>,
        <gasPrice>,
        <gasLimit>,
        <value>,
        "0x <compiled smart contract code>" + encodedConstructor);

// send...
```

Transacting with a smart contract

To transact with an existing smart contract, the following attributes will need to be provided:

to the smart contract address

value the amount of Ether you wish to deposit in the smart contract (assumes zero if not provided)

data the encoded function selector and parameter arguments

web3j takes care of the function encoding for you, further details are available in the Ethereum Contract ABI section of the Ethereum Wiki.

```
Function function = new Function<>(
            "functionName",  // function we're calling
            Arrays.asList(new Type(value), ...),  // Parameters to pass as Solidity Types
            Arrays.asList(new TypeReference<Type>() {}, ...));

String encodedFunction = FunctionEncoder.encode(function)
Transaction transaction = Transaction.createFunctionCallTransaction(
            <from>, <gasPrice>, <gasLimit>, contractAddress, <funds>, encodedFunction);

org.web3j.protocol.core.methods.response.EthSendTransaction transactionResponse =
            web3j.ethSendTransaction(transaction).sendAsync().get();

String transactionHash = transactionResponse.getTransactionHash();

// wait for response using EthGetTransactionReceipt...
```

It is not possible to return values from transactional functional calls, regardless of the return type of the message signature. However, it is possible to capture values returned by functions using filters. Please refer to the Filters and Events section for details.

Querying the state of a smart contract

This functionality is facilitated by the eth_call JSON-RPC call.

eth_call allows you to call a method on a smart contract to query a value. There is no transaction cost associated with this function, this is because it does not change the state of any smart contract method's called, it simply returns the value from them:

```
Function function = new Function<>(
            "functionName",
            Arrays.asList(new Type(value)),  // Solidity Types in smart contract functions
            Arrays.asList(new TypeReference<Type>() {}, ...));

String encodedFunction = FunctionEncoder.encode(function)
org.web3j.protocol.core.methods.response.EthCall response = web3j.ethCall(
            Transaction.createEthCallTransaction(contractAddress, encodedFunction),
            DefaultBlockParameterName.LATEST)
            .sendAsync().get();

List<Type> someTypes = FunctionReturnDecoder.decode(
            response.getValue(), function.getOutputParameters());
```

Note: If an invalid function call is made, or a null result is obtained, the return value will be an instance of Collections.emptyList()

Smart Contracts

Developers have the choice of three languages for writing smart contracts:

Solidity The flagship language of Ethereum, and most popular language for smart contracts.

Serpent A Python like language for writing smart contracts.

LISP Like Language (LLL) A low level language, Serpent provides a superset of LLL. There's not a great deal of information for working with LLL, the following blog /var/log/syrinx and associated GitHub is a good place to start.

In order to deploy a smart contract onto the Ethereum blockchain, it must first be compiled into a bytecode format, then it can be sent as part of a transaction request as detailed in *Creation of a smart contract*.

Given that Solidity is the language of choice for writing smart contracts, it is the language supported by web3j, and is used for all subsequent examples.

Getting started with Solidity

An overview of Solidity is beyond the scope of these docs, however, the following resources are a good place to start:

- Contract Tutorial on the Go Ethereum Wiki
- Introduction to Smart Contracts in the Solidity project documentation
- Writing a contract in the Ethereum Homestead Guide

Compiling Solidity source code

Compilation to bytecode is performed by the Solidity compiler, *solc*. You can install the compiler, locally following the instructions as per the project documentation.

To compile the solidity code run:

```
$ solc <contract>.sol --bin --abi --optimize -o <output-dir>/
```

Note: there are issues with installing solc via homebrew on OS X currently, please see the following thread for further information.

The *–bin* and *–abi* compiler arguments are both required to take full advantage of working with smart contracts from web3j.

–bin Outputs a Solidity binary file containing the hex-encoded binary to provide with the transaction request (as per *Creation of a smart contract*).

–abi Outputs a solidity application binary interface (ABI) file which details all of the publicly acessible contract methods and their associated parameters. These details along with the contract address are crucial for interacting with smart contracts. The ABI file is also used for the generation of *Solidity smart contract wrappers*.

There is also a *–gas* argument for providing estimates of the *Gas* required to create a contract and transact with its methods.

Alternatively, you can write and compile solidity code in your browser via the Browser-Solidity. Browser-Solidity is great for smaller smart contracts, but you may run into issues working with larger contracts.

You can also compile solidity code via Ethereum clients such as Geth and Parity, using the JSON-RPC method eth_compileSolidity which is also supported in web3j. However, the Solidity compiler must be installed on the client for this to work.

There are further options available, please refer to the relevant section in the Homestead documentation.

Deploying and interacting with smart contracts

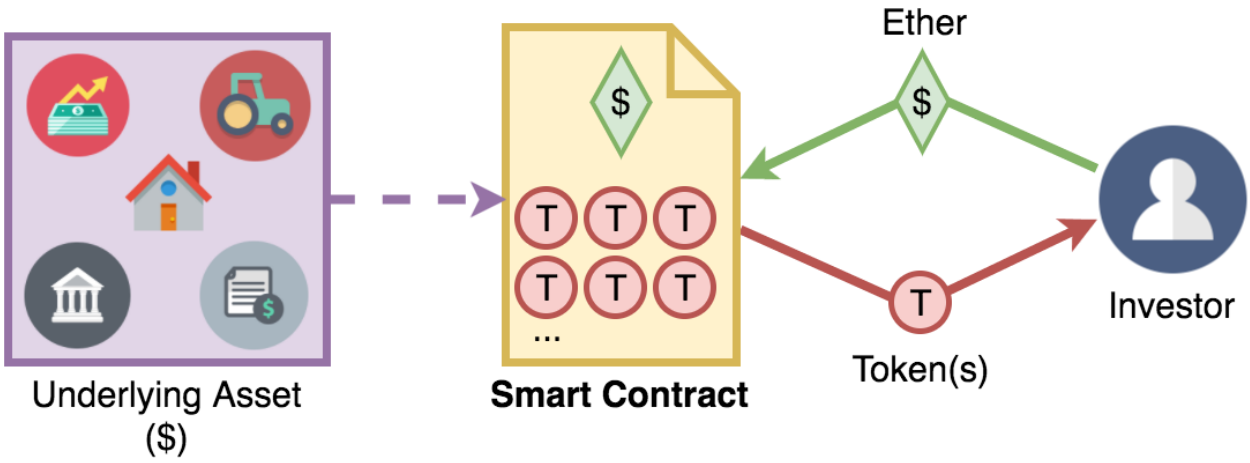
If you want to avoid the underlying implementation detail for working with smart contracts, web3j provides *Solidity smart contract wrappers* which enable you to interact directly with all of a smart contract's methods via a generated wrapper object.

Alternatively, if you wish to send regular transactions or have more control over your interactions with your smart contracts, please refer to the sections *Creation of a smart contract*, *Transacting with a smart contract* and *Querying the state of a smart contract* for details.

Smart contract examples

web3j provides a number of smart contract examples in the project directory src/test/resources/solidity

It also provides integration tests for demonstrating the deploying and working with those smart contracts in the project directory src/integration-test/java/org/web3j/protocol/scenarios.



EIP-20 Ethereum token standard smart contract

There is an active Ethereum Improvement Proposal (EIP), EIP-20 that has been created to define the standard functions that a smart contract providing tokens can implement.

The EIP-20 proposal provides function definitions, but does not provide an implementation example. However, there is an implementation provided in src/test/resources/solidity/contracts, which has been taken from Consensys' Tokens project.

There are two integration tests that have been written to fully demonstrate the functionality of this token smart contract.

HumanStandardTokenGeneratedIT uses the generated HumanStandardTokenGenerated *smart contract wrapper* to demonstrate this.

Alternatively, if you do not wish to use a smart contract wrapper and would like to work directly with the JSON-RPC calls, please refer to HumanStandardTokenIT.

Solidity smart contract wrappers

web3j supports the auto-generation of smart contract function wrappers in Java from Solidity ABI files.

The web3j Command Line Tools tools ship with a command line utility for generating the smart contract function wrappers:

```
$ web3j solidity generate /path/to/<smart-contract>.bin /path/to/<smart-contract>.abi -o /path/to/src
```

Or by calling the Java class directly:

```
org.web3j.codegen.SolidityFunctionWrapperGenerator /path/to/<smart-contract>.bin /path/to/<smart-cont
```

Where the *bin* and *abi* are obtained as per *Compiling Solidity source code*.

The smart contract wrappers support all common operations for working with smart contracts:

- *Construction and deployment*

- *Invoking transactions and events*
- *Calling constant methods*
- *Examples*

Any method calls that requires an underlying JSON-RPC call to take place will return a Future to avoid blocking.

Construction and deployment

Construction and deployment of smart contracts happens with the *deploy* method:

```
YourSmartContract contract = YourSmartContract.deploy(
        <web3j>, <credentials>, GAS_PRICE, GAS_LIMIT,
        <initialValue>,
        <param1>, ..., <paramN>);
```

This will create a new instance of the smart contract on the Ethereum blockchain using the supplied credentials, and constructor parameter values.

It returns a new smart contract wrapper instance which contains the underlying address of the smart contract. If you wish to construct an instance of a smart contract wrapper with an existing smart contract, simply pass in it's address:

```
YourSmartContract contract = YourSmartContract.load(
        "0x<address>", web3j, credentials, GAS_PRICE, GAS_LIMIT);
```

Using this method, you may want to ascertain that the contract address that you have loaded is the smart contract that you expect. For this you can use the isValid smart contract method, which will only return true if the deployed bytecode at the contract address matches the bytecode in the smart contract wrapper.

> **contract.isValid(); // returns false if the contract bytecode does not match what's deployed** // at the provided addres

Transaction Managers

web3j provides a TransactionManager abstraction to control the manner you connect to Ethereum clients with. The default mechanism uses web3j's RawTransactionManager which works with Ethereum wallet files to sign transactions offline before submitting them to the network.

However, you may wish to modify the transaction manager, which you can pass to the smart contract deployment and creation methods instead of a credentials object, i.e.:

```
YourSmartContract contract = YourSmartContract.deploy(
        <web3j>, <transactionManager>, GAS_PRICE, GAS_LIMIT,
        <initialValue>,
        <param1>, ..., <paramN>);
```

In addition to the RawTransactionManager, web3j provides a ClientTransactionManager which passes the responsibility of signing your transaction on to the Ethereum client you are connecting to.

Specifying the Chain Id on Transactions (EIP-155)

The RawTransactionManager takes an optional *chainId* parameter to specify the chain id to be used on transactions as per EIP-155. This prevents transactions from one chain being re-broadcast onto another chain, such as from Morden to Mainnet:

```
TransactionManager transactionManager = new RawTransactionManager(
        web3j, credentials, ChainId.MAIN_NET);
```

In order to avoid having to change config or code to specify which chain you are working with, web3j's default behaviour is to not specify chain ids on transactions to simplify working with the library. However, the recommendation of the Ethereum community is to use them.

Invoking transactions and events

All transactional smart contract methods are named identically to their Solidity methods, taking the same parameter values. Transactional calls do not return any values, regardless of the return type specified on the method. Hence, for all transactional methods the Transaction Receipt associated with the transaction is returned.:

```
TransactionReceipt transactionReceipt = contract.someMethod(
             new Type(...),
             ...).get();
```

The transaction receipt is useful for two reasons:

1. It provides details of the mined block that the transaction resides in
2. Solidity events that are called will be logged as part of the transaction, which can then be extracted

Any events defined within a smart contract will be represented in the smart contract wrapper with a method named *process<Event Name>Event*, which takes the Transaction Receipt and from this extracts the indexed and non-indexed event parameters, which are returned decoded in an instance of the EventValues object.:

```
EventValues eventValues = contract.processSomeEvent(transactionReceipt);
```

Alternatively you can use an Observable filter instead which will listen for events associated with the smart contract:

```
contract.someEventObservable(startBlock, endBlock).
        .subscribe(event -> ...);
```

For more information on working with Observable filters, refer to Filters and Events.

Remember that for any indexed array, bytes and string Solidity parameter types, a Keccak-256 hash of their values will be returned, see the documentation for further information.

Calling constant methods

Constant methods are those that read a value in a smart contract, and do not alter the state of the smart contract. These methods are available with the same method signature as the smart contract they were generated from, the only addition is that the call is wrapped in a Future.:

```
Type result = contract.someMethod(new Type(...), ...).get();
```

Examples

Please refer to *EIP-20 Ethereum token standard smart contract*.

Filters and Events

Filters provide notifications of certain events taking place in the Ethereum network. There are three classes of filter supported in Ethereum:

1. Block filters
2. Pending transaction filters
3. Topic filters

Block filters and pending transaction filters provide notification of the creation of new transactions or blocks on the network.

Topic filters are more flexible. These allow you to create a filter based on specific criteria that you provide.

Unfortunately, unless you are using a WebSocket connection to Geth, working with filters via the JSON-RPC API is a tedious process, where you need to poll the Ethereum client in order to find out if there are any updates to your filters due to the synchronous nature of HTTP and IPC requests. Additionally the block and transaction filters only provide the transaction or block hash, so a further request is required to obtain the actual transaction or block referred to by the hash.

web3j's managed Filter implementation address these issues, so you have a fully asynchronous event based API for working with filters. It uses RxJava's Observables which provides a consistent API for working with events, which facilitates the chaining together of JSON-RPC calls via functional composition.

Note: filters are not supported on Infura.

Block and transaction filters

To receive all new blocks as they are added to the blockchain (the false parameter specifies that we only want the blocks, not the embedded transactions too):

```
Subscription subscription = web3j.blockObservable(false).subscribe(block -> {
    ...
});
```

To receive all new transactions as they are added to the blockchain:

```
Subscription subscription = web3j.transactionObservable().subscribe(tx -> {
    ...
});
```

To receive all pending transactions as they are submitted to the network (i.e. before they have been grouped into a block together):

```
Subscription subscription = web3j.pendingTransactionObservable().subscribe(tx -> {
    ...
});
```

Subscriptions should always be cancelled when no longer required via *unsubscribe*:

```
subscription.unsubscribe();
```

Other callbacks are also provided which provide simply the block or transaction hashes, for details of these refer to the Web3jRx interface.

Replay filters

web3j also provides filters for replaying block and transaction history.

To replay a range of blocks from the blockchain:

```
Subscription subscription = web3j.replayBlocksObservable(
        <startBlockNumber>, <endBlockNumber>, <fullTxObjects>)
        .subscribe(block -> {
            ...
});
```

To replay the individual transactions contained within a range of blocks:

```
Subscription subscription = web3j.replayTransactionsObservable(
        <startBlockNumber>, <endBlockNumber>)
        .subscribe(tx -> {
            ...
});
```

You can also get web3j to replay all blocks up to the most current, and provide notification (via the submitted Observable) once you've caught up:

```
Subscription subscription = web3j.catchUpToLatestBlockObservable(
        <startBlockNumber>, <fullTxObjects>, <onCompleteObservable>)
        .subscribe(block -> {
            ...
});
```

Or, if you'd rather replay all blocks to the most current, then be notified of new subsequent blocks being created:

```
Subscription subscription = web3j.catchUpToLatestAndSubscribeToNewBlocksObservable(
        <startBlockNumber>, <fullTxObjects>)
        .subscribe(block -> {
            ...
});
```

As above, but with transactions contained within blocks:

```
Subscription subscription = web3j.catchUpToLatestAndSubscribeToNewTransactionsObservable(
        <startBlockNumber>)
        .subscribe(tx -> {
            ...
});
```

All of the above filters are exported via the Web3jRx interface.

Topic filters and EVM events

Topic filters capture details of Ethereum Virtual Machine (EVM) events taking place in the network. These events are created by smart contracts and stored in the transaction log associated with a smart contract.

The Solidity documentation provides a good overview of EVM events.

You use the EthFilter type to specify the topics that you wish to apply to the filter. This can include the address of the smart contract you wish to apply the filter to. You can also provide specific topics to filter on. Where the individual topics represent indexed parameters on the smart contract:

```
EthFilter filter = new EthFilter(DefaultBlockParameterName.EARLIEST,
        DefaultBlockParameterName.LATEST, <contract-address>)
             [.addSingleTopic(...) | .addOptionalTopics(..., ...) | ...];
```

This filter can then be created using a similar syntax to the block and transaction filters above:

```
web3j.ethLogObservable(filter).subscribe(log -> {
    ...
});
```

The filter topics can only refer to the indexed Solidity event parameters. It is not possible to filter on the non-indexed event parameters. Additionally, for any indexed event parameters that are variable length array types such as string and bytes, the Keccak-256 hash of their value is stored on the EVM log. It is not possible to store or filter using their full value.

If you create a filter instance with no topics associated with it, all EVM events taking place in the network will be captured by the filter.

A note on functional composition

In addition to *send()* and *sendAsync*, all JSON-RPC method implementations in web3j support the *observable()* method to create an Observable to execute the request asynchronously. This makes it very straight forwards to compose JSON-RPC calls together into new functions.

For instance, the blockObservable is itself composed of a number of separate JSON-RPC calls:

```
public Observable<EthBlock> blockObservable(
        boolean fullTransactionObjects, long pollingInterval) {
    return this.ethBlockHashObservable(pollingInterval)
            .flatMap(blockHash ->
                    web3j.ethGetBlockByHash(blockHash, fullTransactionObjects).observable());
}
```

Here we first create an observable that provides notifications of the block hash of each newly created block. We then use *flatMap* to invoke a call to *ethGetBlockByHash* to obtain the full block details which is what is passed to the subscriber of the observable.

Further examples

Please refer to the integration test ObservableIT for further examples.

For a demonstration of using the manual filter API, you can take a look at the test EventFilterIT..

Command Line Tools

A web3j fat jar is distributed with each release providing command line tools. The command line allow you to use some of the functionality of web3j from your terminal:

These tools provide:

- Wallet creation
- Wallet password management
- Ether transfer from one wallet to another

- Generation of Solidity smart contract wrappers

The command line tools can be obtained as a zipfile/tarball from the releases page of the project repository, under the **Downloads** section, or for OS X users via Homebrew.

```
brew tap web3j/web3j
brew install web3j
```

To run via the zipfile, simply extract the zipfile and run the binary:

```
$ unzip web3j-<version>.zip
   creating: web3j-1.0.2/lib/
  inflating: web3j-1.0.2/lib/core-1.0.2-all.jar
   creating: web3j-1.0.2/bin/
  inflating: web3j-1.0.2/bin/web3j
  inflating: web3j-1.0.2/bin/web3j.bat
$ ./web3j-<version>/bin/web3j


                _       _____ _      _
               | |     |_____ (_)    (_)
__       _____| |__       / /_      _   ___
\ \ /\ / / _ \ '_ \     \ \ |    | | / _ \
 \ V  V /  __/ |_) |.___/ / | _  | || (_) |
  \_/\_/ \___|_.__/ \_____/| |(_)|_| \___/
                          _/ |
                         |__/

Usage: web3j wallet|solidity ...
```

Wallet tools

To generate a new Ethereum wallet:

```
$ web3j wallet create
```

To update the password for an existing wallet:

```
$ web3j wallet update <walletfile>
```

To send Ether to another address:

```
$ web3j wallet send <walletfile> <destination-address>
```

When sending Ether to another address you will be asked a series of questions before the transaction takes place. See the below for a full example

The following example demonstrates using web3j to send Ether to another wallet.

```
$ ./web3j-<version>/bin/web3j wallet send <walletfile> <destination-address>


                _       _____ _      _
               | |     |_____ (_)    (_)
__       _____| |__       / /_      _   ___
\ \ /\ / / _ \ '_ \     \ \ |    | | / _ \
 \ V  V /  __/ |_) |.___/ / | _  | || (_) |
  \_/\_/ \___|_.__/ \_____/| |(_)|_| \___/
                          _/ |
                         |__/
```

```
Please enter your existing wallet file password:
Wallet for address 0x19e03255f667bdfd50a32722df860b1eeaf4d635 loaded
Please confirm address of running Ethereum client you wish to send the transfer request to [http://lo
Connected successfully to client: Geth/v1.4.18-stable-c72f5459/darwin/go1.7.3
What amound would you like to transfer (please enter a numeric value): 0.000001
Please specify the unit (ether, wei, ...) [ether]:
Please confim that you wish to transfer 0.000001 ether (1000000000000 wei) to address 0x9c98e381edc5f
Please type 'yes' to proceed: yes
Commencing transfer (this may take a few minutes).....................................................

Funds have been successfully transferred from 0x19e03255f667bdfd50a32722df860b1eeaf4d635 to 0x9c98e38
Transaction hash: 0xb00afc5c2bb92a76d03e17bd3a0175b80609e877cb124c02d19000d529390530
Mined block number: 1849039
```

Solidity smart contract wrapper generator

Please refer to *Solidity smart contract wrappers*.

Management APIs

In addition to implementing the standard JSON-RPC API, Ethereum clients, such as Geth and Parity provide additional management via JSON-RPC.

One of the key common pieces of functionality that they provide is the ability to create & unlock Ethereum accounts for transacting on the network. In Geth and Parity, this is implemented in their Personal modules, details of which are available below:

- Parity
- Geth

Support for the personal modules is available in web3j. Both clients have further admin commands available, they will be incorporated into web3j over time if there is sufficient demand.

As Parity provides a superset of the personal admin commands offered by Geth, you can initialise a new web3j connector using the Parity factory method:

```
Parity parity = Parity.build(new HttpService());  // defaults to http://localhost:8545/
PersonalUnlockAccount personalUnlockAccount = parity.personalUnlockAccount("0x000...", "a password")
if (personalUnlockAccount.accountUnlocked()) {
    // send a transaction, or use parity.personalSignAndSendTransaction() to do it all in one
}
```

Refer to the integration test ParityIT for further implementation details.

Using Infura with web3j

Signing up

The Infura service by Consensys, provides Ethereum clients running in the cloud, so you don't have to run one yourself to work with Ethereum.

When you sign up to the service you are provided with a token you can use to connect to the relevant Ethereum network:

Main Ethereum Network: https://mainnet.infura.io/your-token

Test Ethereum Network (Ropsten): https://ropsten.infura.io/your-token

If you need some Ether on testnet to get started, please post a message with your wallet address to the web3j Gitter channel and I'll send you some.

InfuraHttpClient

web3j comes with an Infura HTTP client (InfuraHttpService) so you don't need to concern yourself with installing an HTTP certificate into your JVM. Upon startup web3j will connect to the Infura endpoint you have specified and downloads the Infura node TLS certificate to a temporary local key store which exists for the duration of your application runtime:

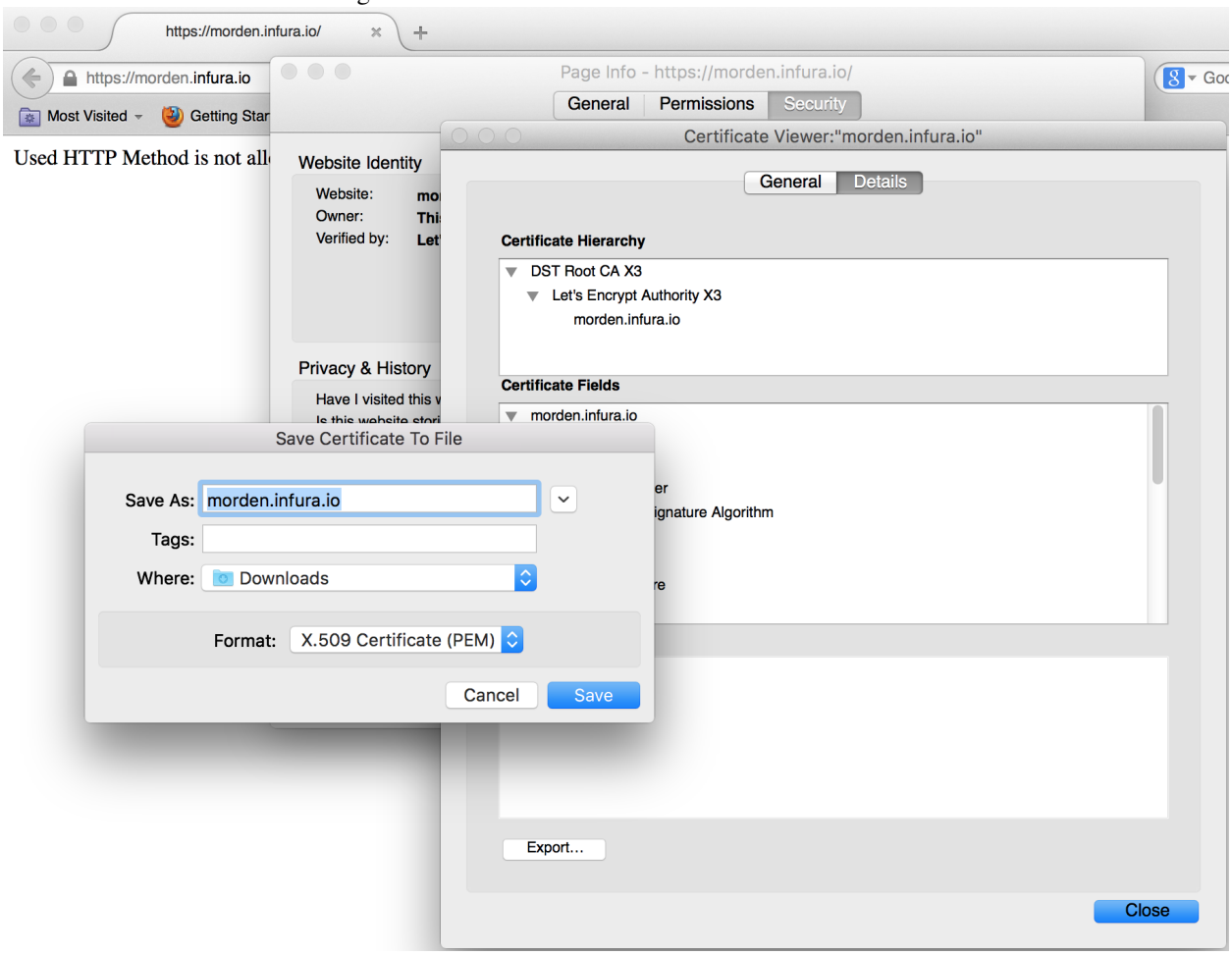
```
Web3j web3 = Web3j.build(new InfuraHttpService("https://morden.infura.io/your-token"));
```

Alternatively, if you'd rather install the key permanently in your JVM keystore, see *below*.

Certificate installation

As the Infura nodes are accessed over TLS, you will need to install the relevant certificates into your Java Virtual Machine's keystore.

You can obtain the certificate using the Firefox web browser:



Alternatively, you can use SSL (this will only download the first certificate in the chain, which should be adequate).

```
$ openssl s_client -connect morden.infura.io:443 -showcerts </dev/null 2>/dev/null|openssl x509 -outi
```

Once you have downloaded it, then install it into your keystore (for Windows/Linux hosts the paths will differ slightly):

```
$ $JAVA_HOME/Contents/Home/jre/bin/keytool -import -noprompt -trustcacerts -alias morden.infura.io -i
```

Connecting

You can now use Infura to work with the Ethereum blockchain.

```
Web3j web3 = Web3j.build(new HttpService("https://morden.infura.io/your-token"));
Web3ClientVersion web3ClientVersion = web3.web3ClientVersion().sendAsync().get();
System.out.println(web3ClientVersion.getWeb3ClientVersion());
```

```
Parity//v1.3.5-beta/x86_64-linux-gnu/rustc1.12.0
```

If you want to test a number of the JSON-RPC calls against Infura, update the integration test CoreIT with your Infura URL & run it.

Transactions

In order to transact with Infura nodes, you will need to create and sign transactions offline before sending them, as Infura nodes have no visibility of your encrypted Ethereum key files, which are required to unlock accounts via the Personal Geth/Parity admin commands.

Refer to the *Offline transaction signing* and Management APIs sections for further details.

Troubleshooting

I'm submitting a transaction, but it's not being mined

After creating and sending a transaction, you receive a transaction hash, however calling eth_getTransactionReceipt always returns a blank value, indicating the transaction has not been mined:

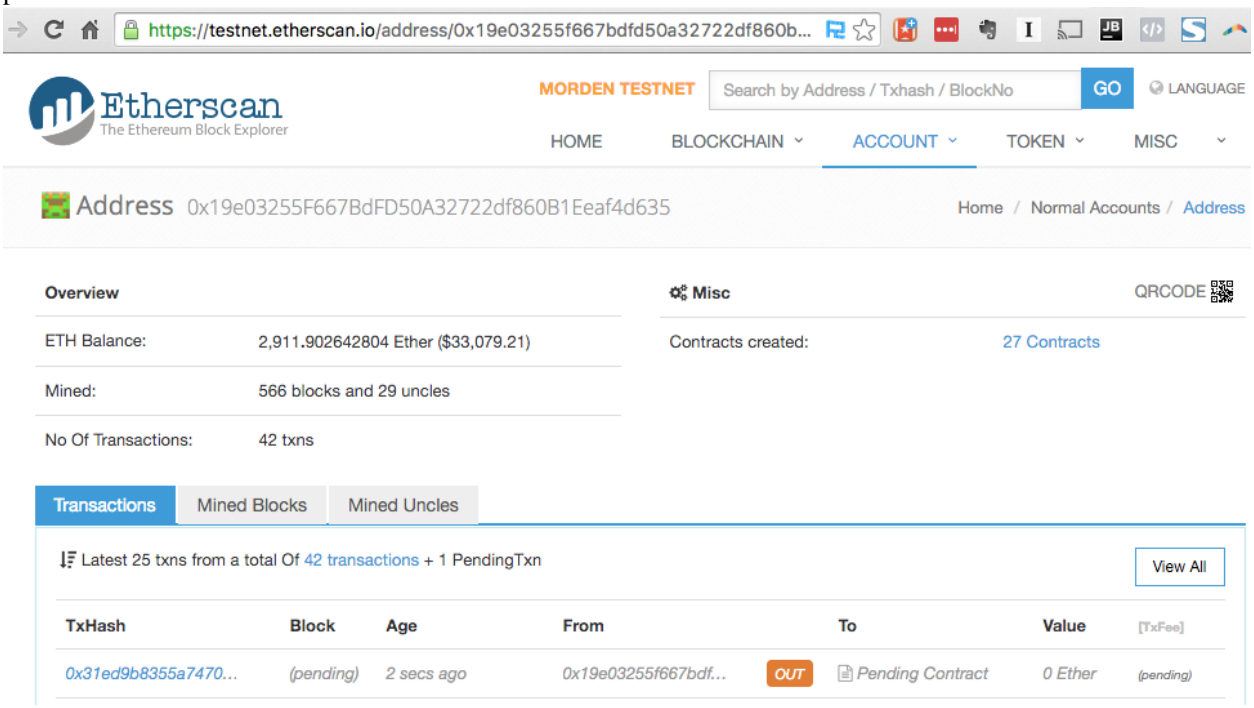
```
String transactionHash = sendTransaction(...);

// you loop through the following expecting to eventually get a receipt once the transaction
// is mined
EthGetTransactionReceipt.TransactionReceipt transactionReceipt =
        web3j.ethGetTransactionReceipt(transactionHash).sendAsync().get();

if (!transactionReceipt.isPresent()) {
    // try again, ad infinitum
}
```

However, you never receive a transaction receipt. Unfortunately there may not be a an error in your Ethereum client indicating any issues with the transaction:

```
I1025 18:13:32.817691 eth/api.go:1185] Tx(0xeaac9aab7f9aeab189acd8714c5a60c7424f86820884b815c4448cfc
```

The easiest way to see if the submission is waiting to mined is to refer to Etherscan and search for the address the transaction was sent using https://testnet.etherscan.io/address/0x... If the submission has been successful it should be

visible in Etherscan within seconds of you performing the transaction submission. The wait is for the mining to take place.



If there is no sign of it then the transaction has vanished into the ether (sorry). The likely cause of this is likely to be to do with the transaction's nonce either not being set, or being too low. Please refer to the section *The transaction nonce* for more information.

I want to see details of the JSON-RPC requests and responses

Set the following system properties in your main class or project configuration:

```
// For HTTP connections
System.setProperty("org.apache.commons.logging.Log","org.apache.commons.logging.impl.SimpleLog");
System.setProperty("org.apache.commons.logging.simplelog.showdatetime", "true");
System.setProperty("org.apache.commons.logging.simplelog.log.org.apache.http.wire", "DEBUG");

// For IPC connections
System.setProperty("org.apache.commons.logging.simplelog.log.org.web3j.protocol.ipc", "DEBUG");
```

I want to obtain some Ether on Testnet, but don't want to have to mine it myself

Head to the Ethereum Ropsten Faucet to request one free Ether.

How do I obtain the return value from a smart contract method invoked by a transaction?

You can't. It is not possible to return values from methods on smart contracts that are called as part of a transaction. If you wish to read a value during a transaction, you must use Events. To query values from smart contracts you must use a call, which is separate to a transaction. These methods should be marked as constant functions. *Solidity smart contract wrappers* created by web3j handle these differences for you.

The following StackExchange post is useful for background.

Is it possible to send arbitrary text with transactions?

Yes it is. Text should be ASCII encoded and provided as a hexadecimal String in the data field of the transaction. This is demonstrated below:

```
RawTransaction.createTransaction(
        <nonce>, GAS_PRICE, GAS_LIMIT, "0x<address>", <amount>, "0x<hex encoded text>");

byte[] signedMessage = TransactionEncoder.signMessage(rawTransaction, ALICE);
String hexValue = Numeric.toHexString(signedMessage);

EthSendTransaction ethSendTransaction =
        parity.ethSendRawTransaction(hexValue).sendAsync().get();
String transactionHash = ethSendTransaction.getTransactionHash();
...
```

Note: Please ensure you increase the gas limit on the transaction to allow for the storage of text.

The following StackExchange post is useful for background.

Do you have a project donation address?

Absolutely, you can contribute Bitcoin or Ether to help fund the development of web3j.

Ethereum	0x2dfBf35bb7c3c0A466A6C48BEBf3eF7576d3C420
Bitcoin	1DfUeRWUy4VjekPmmZUNqCjcJBMwsyp61G

Where can I get commercial support for web3j?

Commercial support and training is available from blk.io.

Projects using web3j

- Ether Wallet by @vikulin
- eth-contract-api by @adridadou
- Ethereum Paper Wallet by @matthiaszimmermann

Developer Guide

Building web3j

web3j includes integration tests for running against a live Ethereum client. If you do not have a client running, you can exclude their execution as per the below instructions.

To run a full build including integration tests:

```
$ ./gradlew check
```

To run excluding integration tests:

```
$ ./gradlew -x integrationTest check
```

Generating documentation

web3j uses the Sphinx documentation generator.

All documentation (apart from the project README.md) resides under the /docs directory.

To build a copy of the documentation, from the project root:

```
$ cd docs
$ make clean html
```

Then browse the build documentation via:

```
$ open build/html/index.html
```

Links and Useful Resources

- Ethereum Homestead Documentation
- Ethereum Wiki
- Ethereum JSON-RPC specification
- Ethereum Yellow Paper and GitHub repository
- Homestead docs
- Solidity docs
- Layout of variables in storage
- Ethereum tests contains lots of common tests for clients
- Etherscan is very useful for exploring blocks and transactions, it also has a testnet site
- Ethstats provides a useful network dashboard. There is also a testnet dashboard, and one for Parity clients.
- Ethereum reddit

Thanks and Credits

- The Nethereum project for the inspiration
- Othera for the great things they are building on the platform
- Finhaus guys for putting me onto Nethereum
- bitcoinj for the reference Elliptic Curve crypto implementation
- Everyone involved in the Ethererum project and its surrounding ecosystem
- And of course the users of the library, who've provided valuable input & feedback - @ice09, @adridadou, @nickmelis, @basavk, @kabl, @MaxBinnewies, @vikulin, @sullis, @vethan, @h2mch, @mtiutin, @fooock, @ermyas, @danieldietrich, @matthiaszimmermann, @ferOnti, @fraspadafora, @bigstar119, @gagarin55, @thedoctor, @tramonex-nate, @ferOnti