CONOR SVENSSON

Blockchain: Using Cryptocurrency with Java

Integrating the Ethereum blockchain into Java apps using web3j

Hardly a day goes by without a mention of blockchain in the technology or financial press. But what's all the fuss about this technology, and how can you work with it from your Java applications? Before I talk about a library, web3j, that makes interaction possible, let me explain what blockchain is and how it works.

A Very Brief History

Blockchain technology started with the cryptocurrency Bitcoin. Bitcoin emerged in 2008, and although it was not the first proposed cryptocurrency, it was the first that was completely decentralized, requiring neither a central authority for issuance nor transaction verification. Bitcoin maintains a distributed ledger containing details of all Bitcoin transactions. All Bitcoin ownership is derived from these ledger entries, known as the Bitcoin blockchain. Transactions on the blockchain are generated using a simple scripting language.

Fast-forward five years later to 2013, when a 19-year-old named Vitalik Buterin started developing a new decentralized platform based on ideas from Bitcoin that provided a more-robust scripting language for the development of applications. That platform, named Ethereum, provided a full Turing-complete language. It was first proposed in early 2014, and it launched in July 2015.

Since then, several other blockchain technologies from different groups have emerged; however, Ethereum is presently the most mature (if you can call it that) by far, and it is the basis of this article.

The Ethereum blockchain is driven by the established cryptocurrency Ether. Ether is the second-largest cryptocurrency after Bitcoin with a market capitalization of approximately US$1 billion dollars, versus Bitcoin's US$10 billion capitalization.

Ethereum is intended to serve the back end of a secure, decentralized internet, where communication is with peers, and you no longer have to interact directly with single entities or organizations. web3j is a lightweight Java library for working with Ethereum, which I use in this article.

What Is a Blockchain?

A blockchain can be thought of as a decentralized, immutable data structure that undergoes state transitions that modify its state. State iterates with transactions or operations on the blockchain, the details of which are written to the blockchain. These transactions are grouped into blocks, which are linked together, making up the blockchain (see Figure 1). Much like the event log used in event sourcing, current state is derived by replaying the state transitions that have taken place previously.

A blockchain is a *distributed ledger technology* (DLT), a term that has emerged for describing technologies such as blockchain that provide decentralized storage of data. Not all DLTs use a common blockchain behind the scenes as in Ethereum; some keep data private between those parties entering into a transaction.

In Ethereum, data is written to, and subsequent state transitions take place within, the Ethereum Virtual Machine (EVM), which—like the Java Virtual Machine—interprets a bytecode format to execute the instructions in a transaction. But unlike the JVM, it uses a network of fully decentralized nodes. Each node executes these instructions locally, verifying that its local version of the blockchain matches those that have been written to the Ethereum blockchain previously. For the addition of a new block (containing new transactions) to be successful, certain nodes must reach a consensus with a majority of nodes on the network regarding what the new state should be. The nodes that create these new blocks are called *miners*.

Mining

The mining nodes use a consensus mechanism called *proof of work* (which incidentally is the same style of consensus mechanism Bitcoin uses). It relies on raw computing power to create new blocks on the blockchain. Blocks are created by continually hashing groups of new transactions that haven't been assigned to a block yet. When a combination of transactions is found that solves a predefined computational problem (referred to as *difficulty*), the miner that found the solution is rewarded by the network with newly minted currency, and that block is then added to the blockchain.

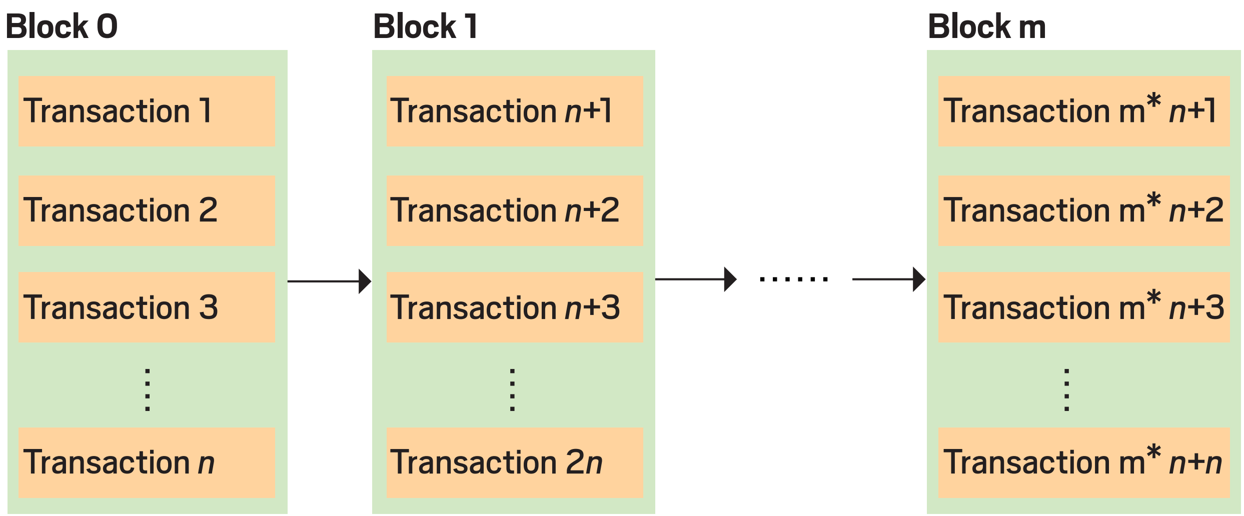


Figure 1. The structure of a blockchain

Smart Contracts

The programs that execute in the EVM are referred to as *smart contracts*. A smart contract is a computerized contract. In Ethereum, smart contracts are a collection of code (functions) and data (state) associated with an address in the Ethereum blockchain.

Getting Started with Ethereum

To start working with the Ethereum network, you need to have access to an Ethereum client (also called a *node* or *peer*). The Ethereum clients synchronize the Ethereum blockchain with one another, and they provide a gateway to interacting with the blockchain.

The two most widely used and complete Ethereum clients available are Geth and Parity.

Simply download one of the clients and then start it up:

```
$ geth --rpcapi personal,db,eth,net,web3 \
    --rpc --testnet

$ parity --chain testnet
```

This will start the client, and it will commence syncing the testnet Ethereum blockchain (which is multiple gigabytes in size). There are two public versions of the Ethereum network: mainnet and testnet (also known as Ropsten), reflecting the live and test environments, respectively. You are going to work with testnet in this article; otherwise, you'll need to start spending real money!

You will also need the latest available version of the web3j command-line tools. These tools provide several useful utility commands for working with Ethereum.

To transact on either the mainnet or testnet, you need to have some Ether cryptocurrency available.

The transaction process relies on public key cryptography, in which users have a public/private key pair. They use the private key (which only they know) to sign a transaction, which is published with the associated public key. The public key can then be used by other network participants to verify that the transaction is authentic, based on the transaction signature.

Fortunately for the users of Ethereum and web3j, the public key cryptography is abstracted away, so you need to work with only an *Ethereum wallet*, a digital file containing account credentials for transacting with Ethereum. A user can have any number of wallet files. It contains a password-encrypted private key and an address that is derived from the public key (note that the public key and address can both be derived from the private key). All transactions on the network are associated with such an address.

You can create a new secure wallet file for Ethereum using the web3j command-line tools, as follows:

```
$ ./web3j-1.0.7/bin/web3j wallet create

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

Please enter a wallet file password:
Please re-enter the password:
Please enter a destination directory location
    [~/.ethereum/testnet-keystore]:
Wallet file <timestamp>--<UUID>.json created in:
~/.ethereum/testnet-keystore
```

Ensure that you select a secure password and that you don't forget it. There is no way of recovering your funds if you forget your password, or if someone steals your wallet file and figures out your password.

Once you have an address for your wallet, you can head over to Etherscan to view the details of the current balance and all transactions associated with this address. Figure 2 shows a wallet address balance on Etherscan.

Testnet has been configured to allow easy mining of Ether, so you can start up a client in mining mode and in a matter of minutes have enough Ether to start creating your own transactions on the network.

Details are available for running a miner with Geth and with Parity.

Getting Started with web3j

With the Ethereum client running, you're now ready to write some Java code to connect to it using web3j. Note that complete listings of all the code are available on GitHub.

The Ethereum clients expose several methods over JSON-RPC, which is a stateless remote procedure call (RPC) protocol using JSON. web3j supports all of the Ethereum JSON-RPC API. However, there is a lot more plumbing behind the scenes that web3j takes care of than just providing JSON-RPC protocol wrappers. The JSON-RPC API is available over HTTP and interprocess communication (IPC), and there is a WebSocket implementation in Geth. However, the most common implementation employs HTTP, which I use in the following examples.

web3j releases are published to both Maven Central and JFrog's JCenter. At the time of this writing, the current release is 1.0.9; however, I'd recommend you check the web3j main project page to use the most current release in your project.

Add the relevant dependency to your build file:

Maven:

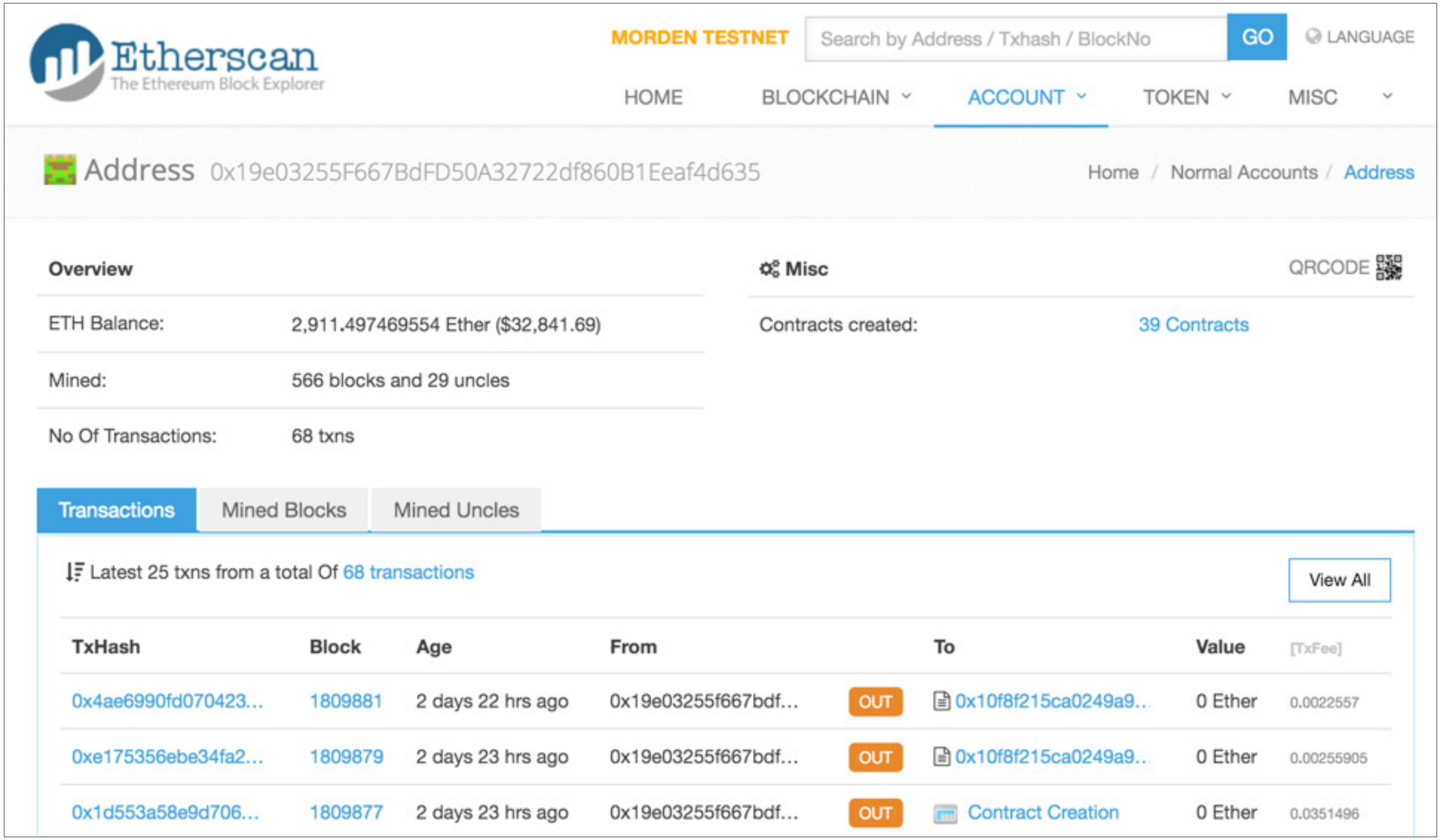
```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>1.0.9</version>
</dependency>
```

Gradle:

```
compile ('org.web3j:core:1.0.9')
```

Place the following code in a runnable class, which displays the Ethereum client version:

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



Address 0x19e03255F667BdFD50A32722df860B1Eeaf4d635

Overview	
ETH Balance:	2,911.497469554 Ether ($32,841.69)
Mined:	566 blocks and 29 uncles
No Of Transactions:	68 txns

Misc	
Contracts created:	39 Contracts

Latest 25 txns from a total Of 68 transactions

TxHash	Block	Age	From		To	Value	[TxFee]
0x4ae6990fd070423...	1809881	2 days 22 hrs ago	0x19e03255f667bdf...	OUT	0x10f8f215ca0249a9...	0 Ether	0.0022557
0xe175356ebe34fa2...	1809879	2 days 23 hrs ago	0x19e03255f667bdf...	OUT	0x10f8f215ca0249a9...	0 Ether	0.00255905
0x1d553a58e9d706...	1809877	2 days 23 hrs ago	0x19e03255f667bdf...	OUT	Contract Creation	0 Ether	0.0351496

Figure 2. Ethereum wallet address balance on Etherscan

```
Web3ClientVersion clientVersion =
  web3.web3ClientVersion().sendAsync().get();

if (!clientVersion.hasError()) {
  System.out.println("Client is running version: " +
      clientVersion.getWeb3ClientVersion());
}
```

When you run this code, if all is well, you should see details of your Ethereum client version:

```
Client is running version:
    Geth/v1.5.4-stable-b70acf3c/darwin/go1.7.3
```

All JSON-RPC method implementations in web3j use the following structure, where web3 is the instantiated web3j implementation (which manages the requests):

```
web3.<method name>([param1, …, paramN])
    .[send()|sendAsync()]
```

The method names and parameters are according to the JSON-RPC specification. The transmission of requests can be done either synchronously or asynchronously via the send() and sendAsync() methods, respectively.

Now that the connectivity to the Ethereum client has been verified, you're ready to start working with the Ethereum blockchain.

Transactions on the Ethereum Blockchain

To create a new transaction on the Ethereum blockchain, you typically perform one of three actions:

- Transferring Ether from one account to another
- Deploying a new smart contract
- Issuing a method call to an existing smart contract that modifies state

There is also a separate read-only method call to examine an existing smart contract, which does not create a transaction on the blockchain.

These transaction interactions require multiple underlying calls via JSON-RPC to Ethereum clients. web3j takes care of this lower-level functionality; however, all JSON-RPC calls are available to users in case they wish to have their own implementation.

Transferring Ether

I'll start by demonstrating the most basic transaction type: transferring 0.2 Ether from one account to another. Make sure that you know the location of the Ethereum wallet file you created earlier, because you won't be able to send any funds without it.

```
Web3j web3 = Web3j.build(new HttpService());

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

TransactionReceipt transactionReceipt =
  Transfer.sendFundsAsync(
      web3, credentials,
      "0x...", BigDecimal.valueOf(0.2),
      Convert.Unit.ETHER).get();

System.out.println(
  "Funds transfer completed, transaction hash: " +
  transactionReceipt.getTransactionHash() +
  ", block number: " +
  transactionReceipt.getBlockNumber());
```

This code produces the following output (the transaction hash has been trimmed and lines have been wrapped for legibility):

```
Funds transfer completed, transaction hash:
0x16e41aa9d97d1c3374a...34,
block number: 1840479
```

The transaction and block hashes are your identifiers for the transaction on the Ethereum blockchain. Behind the scenes, the actual transaction process, illustrated in **Figure 3**, is as follows:

1. The transfer Ether request is submitted to web3j.
2. A transaction message is submitted to an Ethereum client.
3. The client verifies the transaction, and then:
   a. Propagates the transaction on to other Ethereum nodes
   b. Takes a hash of the submitted transaction and sends this to the client in a synchronous HTTP response

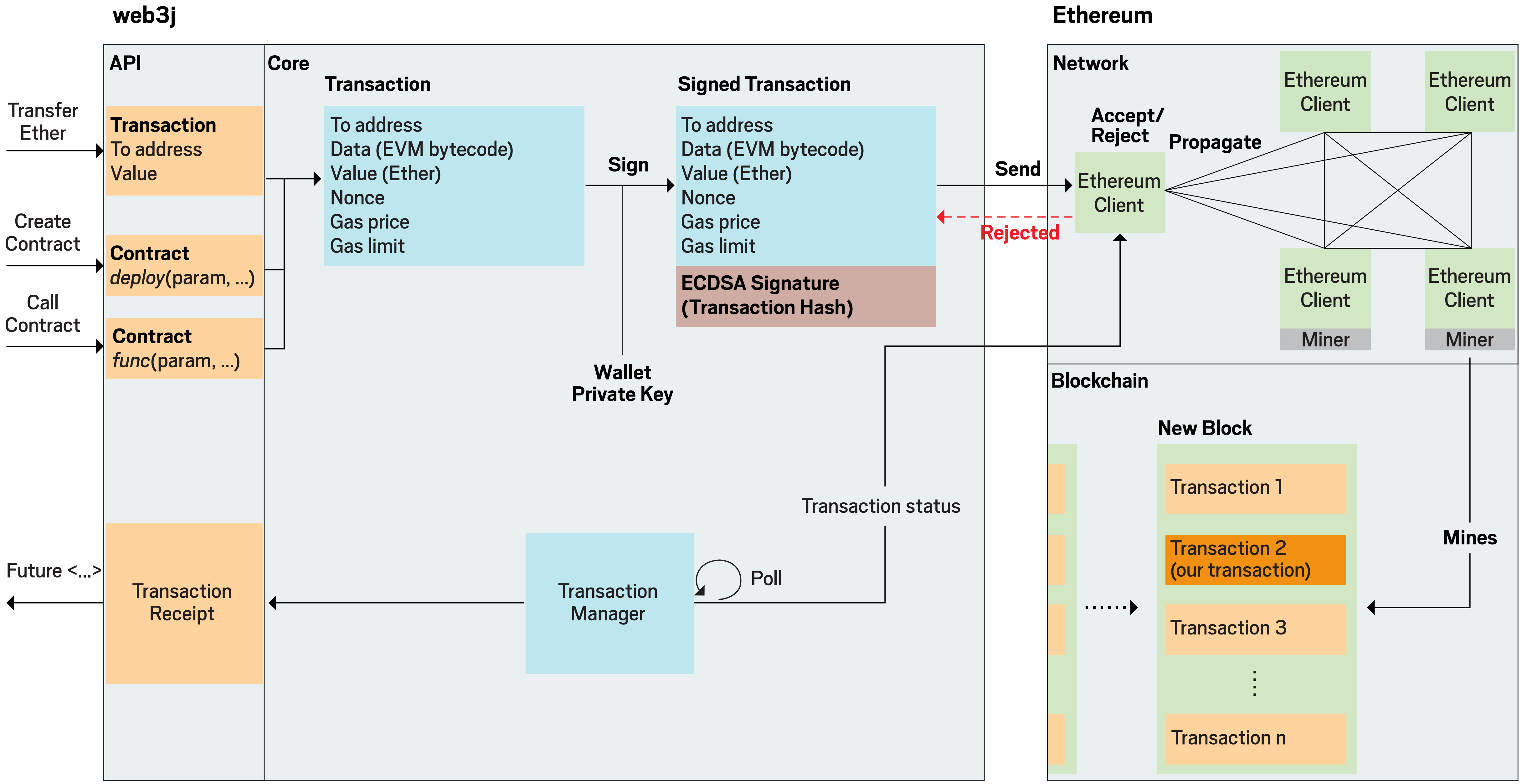
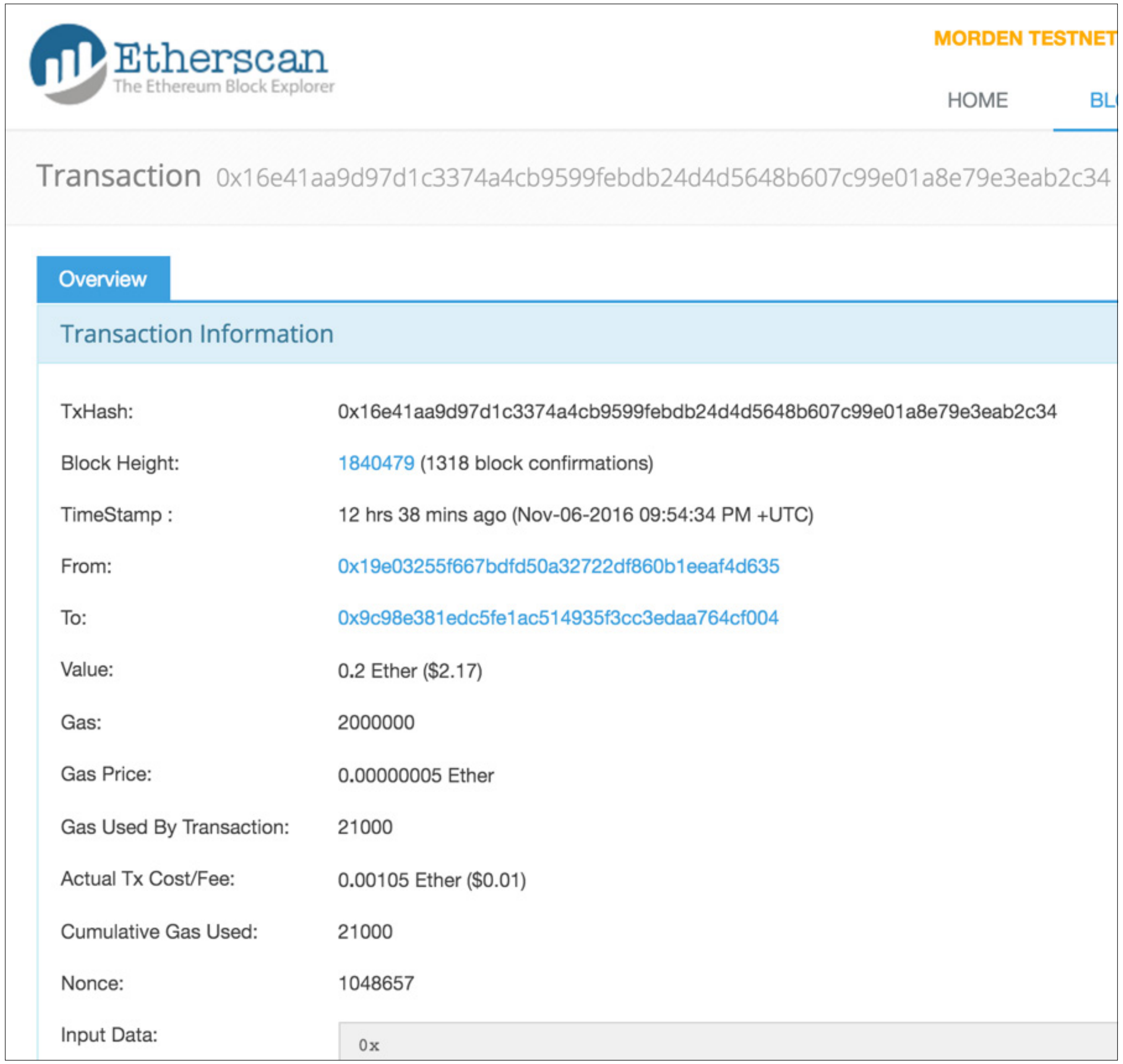


Figure 3. Transaction on Ethereum via web3j

4. This transaction is combined with other new transactions to form a block by miners on the Ethereum network. Once a valid block is formed, the block and details of its associated transactions are immortalized on the blockchain.

You can head back over to Etherscan to view the details of your transaction (**Figure 4**).

Figure 5 show the contents of the block on the blockchain in which the transaction resides.



Transaction 0x16e41aa9d97d1c3374a4cb9599febdb24d4d5648b607c99e01a8e79e3eab2c34

Transaction Information

TxHash:	0x16e41aa9d97d1c3374a4cb9599febdb24d4d5648b607c99e01a8e79e3eab2c34
Block Height:	1840479 (1318 block confirmations)
TimeStamp :	12 hrs 38 mins ago (Nov-06-2016 09:54:34 PM +UTC)
From:	0x19e03255f667bdfd50a32722df860b1eeaf4d635
To:	0x9c98e381edc5fe1ac514935f3cc3edaa764cf004
Value:	0.2 Ether ($2.17)
Gas:	2000000
Gas Price:	0.00000005 Ether
Gas Used By Transaction:	21000
Actual Tx Cost/Fee:	0.00105 Ether ($0.01)
Cumulative Gas Used:	21000
Nonce:	1048657
Input Data:	0x

Figure 4. A receipt of the transaction

Gas

I touched earlier in the article on Ether, which is used to pay for the execution of code in the EVM. There are two parameters that need to be specified with respect to the cost you are prepared to pay for transactions: the gas price and the gas limit. The gas price is the price in Ether you are prepared to pay per gas unit. Each EVM opcode contains a gas cost associated with it. The gas limit is the total amount of gas usage you are willing to pay for the transaction execution. This ensures that all transactions have a finite cost of execution. Details about the gas associated with a transaction are visible in the transaction receipt and Etherscan.

Hello (Ethereum) World

Now that I've demonstrated a simple transaction, things will start to get interesting as I create the first smart contract. Ethereum smart contracts are usually written in a language named Solidity, which is a statically typed high-level language. Describing how to use Solidity could fill many articles, so I'm going to keep the example simple. You can read more about Solidity online.

Now let's use the Greeter contract example. The Greeter contract is the "hello world" example of a smart contract of Ethereum. When you deploy the contract, you pass a UTF 8-encoded string to its constructor. Then, whenever you call the deployed contract, the value of this string is returned by the node on the Ethereum network that processes your request.

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

    /* this function is executed at initialization
     and sets the owner of the contract */
    function mortal() { owner = msg.sender; }
    /* Function to recover the funds on the
    contract */
    function kill() {
        if (msg.sender == owner) suicide(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;
    }
}
```

Although the language is unfamiliar, the example above is fairly intuitive to follow. The `mortal` contract is the base class, and the `greeter` contract is a child class.

The creator of the contract is the instance variable `owner`, of type `address`; `msg.sender` is the sender of any transactions with the contract; and the instance variable `greeting` is another instance variable.

There are two methods to be concerned with:

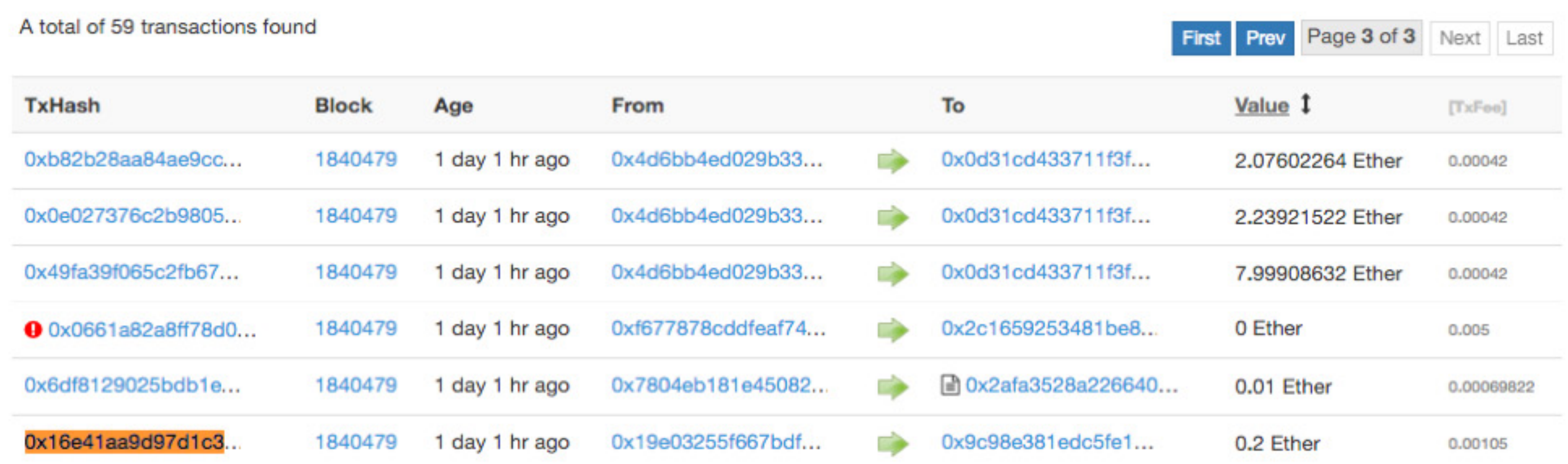
- `greeter(string _greeting)` is the constructor.
- `greet()` is a getter method returning the value of `greeter`.

There are two options for compiling the contract: either install the Solidity compiler or use the browser-based Solidity editor to do it online.

Once the compiler is installed, you can now compile the Greeter contract:

```
$ solc Greeter.sol --bin --abi --optimize -o build/
```

This action creates two files: a binary (.bin) file, which is the smart contract code in a format the EVM can interpret, and an application binary interface (.abi) file, which defines the



A total of 59 transactions found

TxHash	Block	Age	From		To	Value ↕	[TxFee]
0xb82b28aa84ae9cc...	1840479	1 day 1 hr ago	0x4d6bb4ed029b33...	➡	0x0d31cd433711f3f...	2.07602264 Ether	0.00042
0x0e027376c2b9805...	1840479	1 day 1 hr ago	0x4d6bb4ed029b33...	➡	0x0d31cd433711f3f...	2.23921522 Ether	0.00042
0x49fa39f065c2fb67...	1840479	1 day 1 hr ago	0x4d6bb4ed029b33...	➡	0x0d31cd433711f3f...	7.99908632 Ether	0.00042
❗ 0x0661a82a8ff78d0...	1840479	1 day 1 hr ago	0xf677878cddfeaf74...	➡	0x2c1659253481be8...	0 Ether	0.005
0x6df8129025bdb1e...	1840479	1 day 1 hr ago	0x7804eb181e45082...	➡	🗎 0x2afa3528a226640...	0.01 Ether	0.00069822
0x16e41aa9d97d1c3...	1840479	1 day 1 hr ago	0x19e03255f667bdf...	➡	0x9c98e381edc5fe1...	0.2 Ether	0.00105

Figure 5. The block containing the transaction

public methods available on the smart contract.

web3j requires these two files in order to generate the smart contract wrappers for working with the smart contract in Java.

I generate the greeter wrappers using the web3j command-line tools, this time with the solidity command (full paths have been shortened for brevity):

```
$ ./web3j-1.0.9/bin/web3j solidity generate \
build/greeter.bin build/greeter.abi \
-p org.web3j.javamag.generated -o src/main/java/
```

This command will create the class file `org.web3j.example.generated.Greeter`, which wraps all the smart contracts' methods so they can be called from Java:

```
public final class Greeter extends Contract {
 private static final String BINARY =
   "6060604052604051610269380380610269833981...";

 private Greeter(
  String contractAddress, Web3j web3j,
  Credentials credentials,
  BigInteger gasPrice, BigInteger gasLimit) {
  super(
    contractAddress, web3j, credentials,
    gasPrice, gasLimit);
 }

...

 public Future<Utf8String> greet() {
   Function function = new Function("greet",
     Arrays.<Type>asList(),
     Arrays.<TypeReference<?>>asList(
       new TypeReference<Utf8String>() {}));
  return executeCallSingleValueReturnAsync(
    function);
 }

 public static Future<Greeter> deploy(
   Web3j web3j, Credentials credentials,
   BigInteger gasPrice, BigInteger gasLimit,
   BigInteger initialValue,
   Utf8String _greeting) {
  String encodedConstructor =
    FunctionEncoder.encodeConstructor(
      Arrays.<Type>asList(_greeting));
  return deployAsync(
    Greeter.class, web3j, credentials,
    gasPrice, gasLimit,
    BINARY, encodedConstructor, initialValue);
 }

 public static Greeter load(
   String contractAddress, Web3j web3j,
   Credentials credentials,
   BigInteger gasPrice, BigInteger gasLimit) {
  return new Greeter(
    contractAddress, web3j, credentials,
    gasPrice, gasLimit);
 }
}
```

I can now both deploy and call the smart contract:

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

Greeter contract = Greeter.deploy(
  web3, credentials, BigInteger.ZERO,
    new Utf8String("Hello blockchain world!"))
    .get();

Utf8String greeting = contract.greet().get();
System.out.println(greeting.getTypeAsString());
```

Running this code should produce the following output:

```
Hello blockchain world!
```

Now let's run through a more complex smart contract.

A Quick Note on Solidity Types

Solidity has several different native types. Although they are similar to some of those available in Java, web3j requires you to convert native Java types into Solidity types. This requirement is to guarantee the consistency of data being written to, or read from, the Ethereum blockchain.

It's also worth keeping in mind that the EVM uses unsigned 256-bit integers by default, which is why you'll find yourself working with BigInteger types with web3j.

Issuing Your Own Virtual Tokens

Smart contracts can be used to issue and manage token holdings, where a token is representative of proportional ownership tied to some real asset or even its own virtual currency. For instance, you could have a smart contract representing shared ownership of a property. Investors in this property could be provided with the number of tokens that represent their proportion of ownership of that property.

A standard contract API for tokens has been defined for Ethereum. It defines the following methods for interacting with the smart contract:

```
// total supply of tokens
function totalSupply() constant
    returns (uint256 totalSupply)

// tokens associated with address
function balanceOf(address _owner) constant
    returns (uint256 balance)

// transfer tokens from sender account to address
function transfer(address _to, uint256 _value)
    returns (bool success)

// approve spending for _spender of up to _value of
// your tokens
function approve(address _spender, uint256 _value)
    returns (bool success)

// delegated transfer of tokens by approved spender
function transferFrom(address _from, address _to,
    uint256 _value) returns (bool success)

// get the number of tokens the spender is still
// allowed to make
function allowance(address _owner, address _spender)
    constant returns (uint256 remaining)
```

This smart contract also introduces events, which are used in Ethereum to record specific details during smart contract execution on the blockchain. Because a transaction in Ethereum in a smart contract cannot return a value, these events enable you to query information from transactions that took place.

```
// notification of a transfer of tokens
event Transfer(address indexed _from,
    address indexed _to, uint256 _value)

// notification of an approval of delegated transfer
// of tokens
event Approval(address indexed _owner,
    address indexed _spender, uint256 _value)
```

Consensys has made available a full implementation of this smart contract, which you can download and then compile with the Solidity compiler:

```
$ solc HumanStandardToken.sol --bin --abi \
    --optimize -o build/
```

You can then generate smart contract wrappers for this smart contract, as shown next. (Again, full paths are trimmed for brevity.)

```
$ ./web3j-1.0.7/bin/web3j solidity generate \
build/HumanStandardToken.bin \
build/HumanStandardToken.abi \
-p org.web3j.example.generated -o src/main/java/
```

You're now ready to work with some of your very own tokens, and start issuing them:

```
// deploy your smart contract
HumanStandardToken contract = HumanStandardToken
    .deploy(
        web3, credentials, BigInteger.ZERO,
        new Uint256(BigInteger.valueOf(1000000)),
        new Utf8String("web3j tokens"),
        new Uint8(BigInteger.TEN),
        new Utf8String("w3j$"))
    .get();

// print the total supply issued
Uint256 totalSupply = contract.totalSupply().get();
System.out.println("Token supply issued: " +
    totalSupply.getValue());

// check your token balance
Uint256 balance = contract.balanceOf(
    new Address(credentials.getAddress()))
    .get();
System.out.println("Your current balance is: w3j$" +
    balance.getValue());

// transfer tokens to another address
TransactionReceipt transferReceipt =
    contract.transfer(
        new Address("0x<destination address"),
        new Uint256(BigInteger.valueOf(100))).get();
```

You can refer to this article's accompanying code for the full example.

Conclusion

In this article, I have scratched the surface of working with the Ethereum blockchain. There are many further details I have had to gloss over or omit entirely, but I hope this has given you some appreciation of what this fascinating technology is capable of.

For further information, you can refer to the web3j project source code and the documentation, which provides a lot more background information on Ethereum and web3j than I can fit into a single article. </article>

Conor Svensson (@conors10) is the author of web3j, the Java library for integrating applications with the Ethereum blockchain. He previously cofounded the startups coHome and Huffle. He is currently helping Othera build its blockchain lending platform and exchange. He blogs about technology and finance. When not in front of a screen, Svensson likes to make the most of surfing at his local beach, Maroubra, in Sydney, Australia.