web3j Documentation

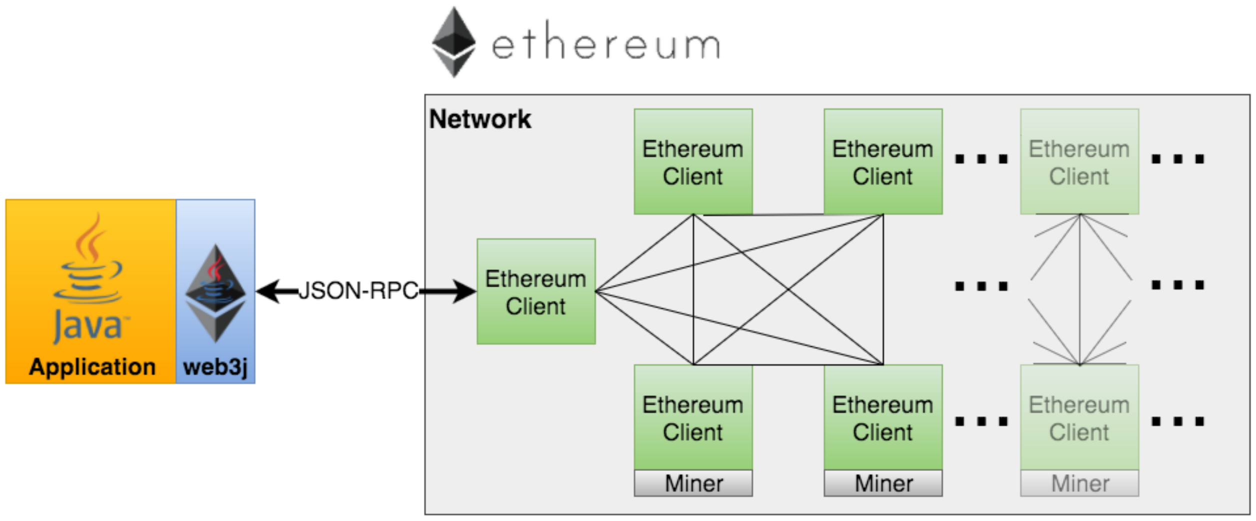
Release 4.1.0

Conor Svensson

March 17, 2019

Contents

web3j is a highly modular, reactive, type safe Java and Android library for working with Smart Contracts and 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
- Auto-generation of Java smart contract wrappers to create, deploy, transact with and call smart contracts from native Java code (Solidity and Truffle definition formats supported)
- Reactive-functional API for working with filters
- Ethereum Name Service (ENS) support
- 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
- Support for ERC20 and ERC721 token standards
- 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 five runtime dependencies:

- RxJava for its reactive-functional API
- OKHttp for HTTP connections
- Jackson Core for fast JSON serialisation/deserialisation
- Bouncy Castle (Spongy Castle on Android) for crypto
- Jnr-unixsocket for *nix IPC (not available on Android)

It also uses JavaPoet for generating smart contract wrappers

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:

5.1 Quickstart

A web3j sample project is available that demonstrates a number of core features of Ethereum with web3j, including:

- Connecting to a node on the Ethereum network
- Loading an Ethereum wallet file
- Sending Ether from one address to another
- Deploying a smart contract to the network
- Reading a value from the deployed smart contract
- Updating a value in the deployed smart contract
- Viewing an event logged by the smart contract

5.2 Getting Started

Add the latest web3j version to your project build configuration.

5.2.1 Maven

Java 8:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>4.2.0</version>
</dependency>
```

Android:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>4.2.0-android</version>
</dependency>
```

5.2.2 Gradle

Java 8:

```
compile ('org.web3j:core:4.2.0')
```

Android:

```
compile ('org.web3j:core:4.2.0-android')
```

5.2.3 Start a client

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

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

Or Parity:

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

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

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

For further information refer to Using Infura with web3j.

Instructions on obtaining Ether to transact on the network can be found in the *testnet section of the docs*.

When you no longer need a *Web3j* instance you need to call the *shutdown* method to close resources used by it.

```
web3.shutdown()
```

5.2.4 Start sending requests

To send synchronous requests:

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

To send asynchronous requests using a CompletableFuture (Future on Android):

```
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 Flowable:

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

5.2.5 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 relevant *IpcService* implementation 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 available on *web3j-android*.

5.2.6 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 the JVM.

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 -b /path/to/<smart-contract>.bin -a /path/to/<smart-contract>.abi -o /path/to
```

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,
        <param1>, ..., <paramN>).send();  // constructor params
```

Or use an existing contract:

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

To transact with a smart contract:

```
TransactionReceipt transactionReceipt = contract.someMethod(
             <param1>,
             ...).send();
```

To call a smart contract:

```
Type result = contract.someMethod(<param1>, ...).send();
```

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

5.2.7 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.blockFlowable(false).subscribe(block -> {
    ...
});
```

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

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

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

Topic filters are also supported:

```
EthFilter filter = new EthFilter(DefaultBlockParameterName.EARLIEST,
        DefaultBlockParameterName.LATEST, <contract-address>)
             .addSingleTopic(...)|.addOptionalTopics(..., ...)|...;
web3j.ethLogFlowable(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.

5.2.8 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<address>|<ensName>",
        BigDecimal.valueOf(1.0), Convert.Unit.ETHER)
        .send();
```

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).send();
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).send();
// ...
```

Although it's far simpler using web3j's Transfer for transacting with Ether.

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

```
Admin web3j = Admin.build(new HttpService());  // defaults to http://localhost:8545/
PersonalUnlockAccount personalUnlockAccount = web3j.personalUnlockAccount("0x000...", "a password").s
if (personalUnlockAccount.accountUnlocked()) {
    // send a transaction
}
```

If you want to make use of Parity's Personal or Trace, or Geth's Personal client APIs, you can use the *org.web3j:parity* and *org.web3j:geth* modules respectively.

5.2.9 Publish/Subscribe (pub/sub)

Ethereum clients implement the pub/sub mechanism that provides the capability to subscribe to events from the network, allowing these clients to take custom actions as needed. In doing so it alleviates the need to use polling and is more efficient. This is achieved by using the WebSocket protocol instead of HTTP protocol.

Pub/Sub methods are available via the *WebSocketService* class, and allows the client to:

- send an RPC call over WebSocket protocol
- subscribe to WebSocket events
- unsubscribe from a stream of events

To create an instance of the *WebSocketService* class you need to first to create an instance of the *WebSocketClient* that connects to an Ethereum client via WebSocket protocol, and then pass it to the *WebSocketService* constructor:

```
final WebSocketClient webSocketClient = new WebSocketClient(new URI("ws://localhost/"));
final boolean includeRawResponses = false;
final WebSocketService webSocketService = new WebSocketService(webSocketClient, includeRawResponses)
```

To send an RPC request using the WebSocket protocol one need to use the *sendAsync* method on the *WebSocketService* instance:

```
// Request to get a version of an Ethereum client
final Request<?, Web3ClientVersion> request = new Request<>(
     // Name of an RPC method to call
     "web3_clientVersion",
     // Parameters for the method. "web3_clientVersion" does not expect any
     Collections.<String>emptyList(),
     // Service that is used to send a request
     webSocketService,
     // Type of an RPC call to get an Ethereum client version
     Web3ClientVersion.class);

// Send an asynchronous request via WebSocket protocol
final CompletableFuture<Web3ClientVersion> reply = webSocketService.sendAsync(
                request,
                Web3ClientVersion.class);

// Get result of the reply
final Web3ClientVersion clientVersion = reply.get();
```

To use synchronous communication (i.e send a request and await a response) one would need to use the *sync* method instead:

```
// Send a (synchronous) request via WebSocket protocol
final Web3ClientVersion clientVersion = webSocketService.send(
                request,
                Web3ClientVersion.class);
```

To subscribe to WebSocket events *WebSocketService* provides the *subscribe* method. *subscribe* returns an instance of the Flowable interface from the RxJava library, which allows the processing of incoming events from an Ethereum network as a reactive stream.

To subscribe to a stream of events you should use *WebSocketService* to send an RPC method via WebSocket; this is usually *eth_subscribe*. Events that it subscribes to depend on parameters to the *eth_subscribe* method. You can find more in the RPC documentation:

```
// A request to subscribe to a stream of events
final Request<?, EthSubscribe> subscribeRequest = new Request<>(
    // RPC method to subscribe to events
    "eth_subscribe",
    // Parameters that specify what events to subscribe to
    Arrays.asList("newHeads", Collections.emptyMap()),
    // Service that is used to send a request
    webSocketService,
    EthSubscribe.class);

final Flowable<NewHeadsNotification> events = webSocketService.subscribe(
     subscribeRequest,
     // RPC method that should be used to unsubscribe from events
     "eth_unsubscribe",
     // Type of events returned by a request
     NewHeadsNotification.class
);

// Subscribe to incoming events and process incoming events
final Disposable disposable = events.subscribe(event -> {
    // Process new heads event
});
```

Notice that we need to provide a name of a method to *WebSocketService* that needs to be called to unsubscribe from a stream of events. This is because different Ethereum clients may have different methods to unsubscribe from particular events. For example, the Parity client requires use of the *parity_unsubscribe* method to unsubscribe from pub/sub events.

To unsubscribe from a stream of events one needs to use a *Flowable* instance for a particular events stream:

```
final Flowable<NewHeadsNotification> events = ...
final Disposable disposable = events.subscribe(...)
disposable.dispose();
```

The methods described above are quite low-level, so we can use *Web3j* implementation instead:

```
final WebSocketService webSocketService = ...
final Web3j web3j = Web3j.build(webSocketService)
final Flowable<NewHeadsNotification> notifications = web3j.newHeadsNotifications()
```

5.2.10 Command line tools

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

- Wallet creation
- Wallet password management
- Transfer of funds from one wallet to another
- Generate Solidity smart contract function wrappers

Please refer to the documentation for further information.

5.2.11 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.
- Asynchronous requests are wrapped in a Java 8 CompletableFutures. web3j provides a wrapper around all async requests to ensure that any exceptions during execution will be captured rather then silently discarded. This is due to the lack of support in *CompletableFutures* for checked exceptions, which are often rethrown as unchecked exception causing problems with detection. See the Async.run() and its associated test for details.

In both the Java 8 and Android builds:

- Quantity payload types are returned as BigIntegers. For simple results, you can obtain the quantity as a String via Response.getResult().
- It's also possible to include the raw JSON payload in responses via the *includeRawResponse* parameter, present in the HttpService and IpcService classes.

5.3 Modules

To provide greater flexibility for developers wishing to work with web3j, the project is made up of a number of modules.

In dependency order, they are as follows:

- utils - Minimal set of utility classes
- rlp - Recursive Length Prefix (RLP) encoders
- abi - Application Binary Interface (ABI) encoders
- crypto - cryptographic library for for transaction signing and key/wallet management in Ethereum
- tuples - Simple tuples library
- core - Much like the previous web3j core artifact without the code generators
- codegen - code generators
- console - command-line tools

The below modules only depend on the core module.

- geth - Geth specific JSON-RPC module
- parity - Parity specific JSON-RPC module
- infura - Infura specific HTTP header support
- contracts - Support for specific EIP's (Ethereum Improvement Proposals)

For most use cases (interacting with the network and smart contracts) the *core* module should be all you need. The dependencies of the core module are very granular and only likely to be of use if your project is focussed on a very specific interaction with the Ethereum network (such as ABI/RLP encoding, transaction signing but not submission, etc).

All modules are published to both Maven Central and Bintray, with the published artifact names using the names listed above, i.e.:

For Java: org.web3j:<module-name>:<version>

For Android: org.web3j:<module-name>:<version>-android

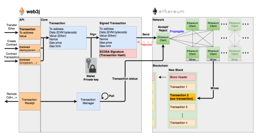
5.4 Transactions

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*.



5.4.1 Obtaining Ether

To obtain Ether you have two options:

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

Mining it yourself in a private environment, or the public test 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.

5.4.2 Ethereum testnets

There are a number of dedicated test networks in Ethereum, which are supported by various clients.

- Rinkeby (Geth only)
- Kovan (Parity only)
- Ropsten (Geth and Parity)

For development, its recommended you use the Rinkeby or Kovan test networks. This is because they use a Proof of Authority (PoA) consensus mechanism, ensuring transactions and blocks are created in a consistent and timely manner. The Ropsten testnet, although closest to the Mainnet as it uses Proof of Work (PoW) consensus, has been subject to attacks in the past and tends to be more problematic for developers.

You can request Ether for the Rinkeby testnet via the Rinkeby Crypto Faucet, available at https://www.rinkeby.io/.

Details of how to request Ether for the Kovan testnet are available here.

If you need some Ether on the Ropsten testnet to get started, please post a message with your wallet address to the web3j Gitter channel and you will be sent some.

5.4.3 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.

However, as mentioned *above* it's simpler to use the Kovan or Rinkeby test networks.

5.4.4 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. web3j uses a default price of 22,000,000,000 Wei (22 x 10^{-8} Ether). This is defined in ManagedTransaction.

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 6,700,000. 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.

5.4.5 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.

5.4.6 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:

```
Admin web3j = Admin.build(new HttpService());
```

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

```
PersonalUnlockAccount personalUnlockAccount = web3j.personalUnlockAccount("0x000...", "a password").s
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>),  // we use default gas limit
              "0x...<smart contract code to execute>"
      );

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

      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.

5.4.7 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.

You can also perform out of process transaction signing if required. This can be achieved by overriding the *sign* method in ECKeyPair.

5.4.8 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.

5.4.9 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 using Recursive Length Prefix encoding
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.

5.4.10 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>);
```

5.4.11 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.

5.4.12 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...
```

However, it is recommended that you use the Transfer class for sending Ether, which takes care of the nonce management and polling for a response for you:

```
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<address>|<ensName>",
        BigDecimal.valueOf(1.0), Convert.Unit.ETHER).send();
```

5.4.13 Recommended approach for working with smart contracts

When working with smart contract wrappers as outlined below, you will have to perform all of the conversions from Solidity to native Java types manually. It is far more effective to use web3j's *Solidity smart contract wrappers* which take care of all code generation and this conversion for you.

5.4.14 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 =
             web3j.ethGetTransactionReceipt(transactionHash).send();

if (transactionReceipt.getTransactionReceipt.isPresent()) {
    String contractAddress = transactionReceipt.get().getContractAddress();
} else {
    // try again
}
```

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...
```

5.4.15 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 (if the smart contract accepts ether)

data the encoded function selector and parameter arguments

web3j takes care of the function encoding for you, for further details on the implementation refer to the Application Binary Interface section.

```
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.

5.4.16 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(<from>, 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()

5.5 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 lll-resurrected GitHub repository 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. web3j can do all of this for you with its *Solidity smart contract wrappers*. To understand what is happening behind the scenes, you can refer to the details 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.

5.5.1 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

5.5.2 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>/
```

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. This is required only for *deploy* and *isValid* *Solidity smart contract wrappers* methods.

–abi Outputs a Solidity Application Binary Interface (ABI) file which details all of the publicly accessible 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 project. 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.

5.5.3 Deploying and interacting with smart contracts

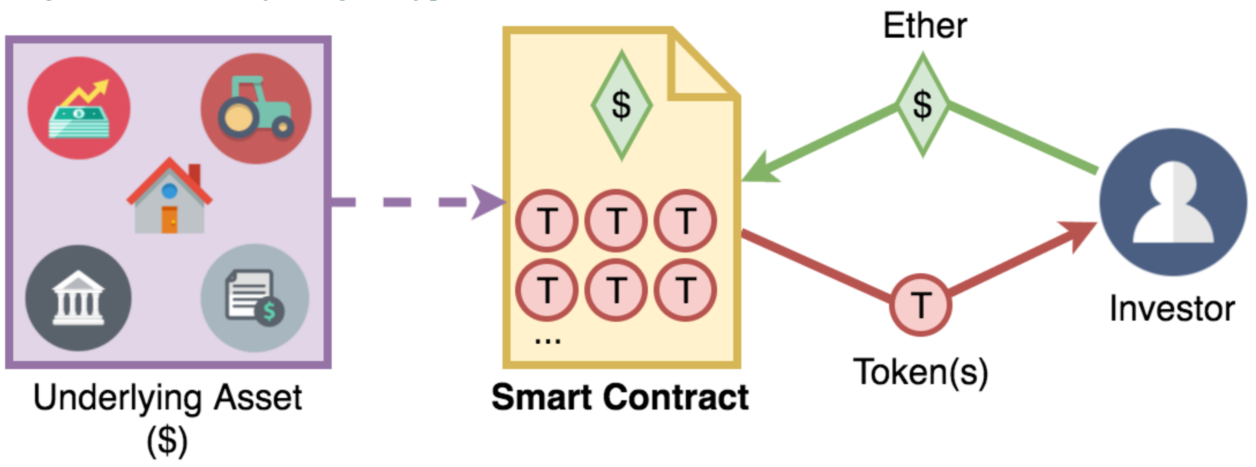
If you want to avoid the underlying implementation detail for working with smart contracts, web3j provides *Solidity smart contract wrappers* which enable you to interact directly with all of a smart contract's methods via a generated wrapper object.

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.

5.5.4 Smart contract examples

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

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



5.5.5 EIP-20 Ethereum token standard smart contract

There an Ethereum standard, EIP-20 which started off as an Ethereum Improvement Proposal (EIP), that defines the standard functions that a smart contract providing tokens should implement.

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

Open Zepplin also provide an example implementation on GitHub.

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 underlying JSON-RPC calls, please refer to HumanStandardTokenIT.

5.5.6 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 [-hV] [-jt] [-st] -a=<abiFile> [-b=<binFile>] -o=<destinationFileDir> -p=<p

   -h, --help                      Show this help message and exit.
   -V, --version                   Print version information and exit.
   -jt, --javaTypes                use native java types. Default: true
   -st, --solidityTypes            use solidity types.
   -a, --abiFile=<abiFile>         abi file with contract definition.
   -b, --binFile=<binFile>         optional bin file with contract compiled code in order to genera
```

```
  -o, --outputDir=<destinationFileDir> destination base directory.
  -p, --package=<packageName>       base package name.
```

BinFile is required for *Contract validity*.

In versions prior to 3.x of web3j, the generated smart contract wrappers used native Solidity types. From web3j 3.x onwards, Java types are created by default. You can create Solidity types using the *–solidityTypes* command line argument.

You can also generate the wrappers by calling the Java class directly:

```
org.web3j.codegen.SolidityFunctionWrapperGenerator -b /path/to/<smart-contract>.bin -a /path/to/<smar
```

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

The native Java to Solidity type conversions used are detailed in the Application Binary Interface section.

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

- *Construction and deployment*
- *Invoking transactions and events*
- *Calling constant methods*
- *Contract validity*

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

web3j also supports the generation of Java smart contract function wrappers directly from Truffle's Contract Schema via the Command Line Tools utility.

```
$ web3j truffle generate [--javaTypes|--solidityTypes] /path/to/<truffle-smart-contract-output>.json
```

And this also can be invoked by calling the Java class:

```
org.web3j.codegen.TruffleJsonFunctionWrapperGenerator /path/to/<truffle-smart-contract-output>.json -
```

A wrapper generated this way ia "enhanced" to expose the per-network deployed address of the contract. These addresses are from the truffle deployment at the time the wrapper is generared.

5.5.7 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>).send();
```

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

The *<initialValue>* parameter is only required if your smart contract accepts Ether on construction. This requires the Solidity payable modifier to be present in the contract.

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>|<ensName>", web3j, credentials, GAS_PRICE, GAS_LIMIT);
```

5.5.8 Contract validity

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 address
```

Note: Contract wrapper has to be generated with *–bin* for this to work.

5.5.9 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,
        <param1>, ..., <paramN>).send();
```

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.

There is also a ReadonlyTransactionManager for when you only want to retrieve data from a smart contract, but not transact with it.

5.5.10 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 Ropsten to Mainnet:

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

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.

You can obtain the chain id of the network that your Ethereum client is connected to with the following request:

```
web3j.netVersion().send().getNetVersion();
```

5.5.11 Transaction Receipt Processors

By default, when a new transaction is submitted by web3j to an Ethereum client, web3j will continually poll the client until it receives a TransactionReceipt, indicating that the transaction has been added to the blockchain. If you are sending a number of transactions asynchronously with web3j, this can result in a number of threads polling the client concurrently.

To reduce this polling overhead, web3j provides configurable TransactionReceiptProcessors.

There are a number of processors provided in web3j:

- PollingTransactionReceiptProcessor is the default processor used in web3j, which polls periodically for a transaction receipt for each individual pending transaction.
- QueuingTransactionReceiptProcessor has an internal queue of all pending transactions. It contains a worker that runs periodically to query if a transaction receipt is available yet. If a receipt is found, a callback to the client is invoked.
- NoOpProcessor provides an EmptyTransactionReceipt to clients which only contains the transaction hash. This is for clients who do not want web3j to perform any polling for a transaction receipt.

Note: the EmptyTransactionReceipt is also provided in the the initial response from the QueuingTransactionReceipt-Processor. This allows the caller to have the transaction hash for the transaction that was submitted to the network.

If you do not wish to use the default processor (PollingTransactionReceiptProcessor), you can specify the transaction receipt processor to use as follows:

```
TransactionReceiptProcessor transactionReceiptProcessor =
        new QueuingTransactionReceiptProcessor(web3j, new Callback() {
                 @Override
                 public void accept(TransactionReceipt transactionReceipt) {
                     // process transactionReceipt
                 }

                 @Override
                 public void exception(Exception exception) {
                     // handle exception
                 }
TransactionManager transactionManager = new RawTransactionManager(
        web3j, credentials, ChainId.MAINNET, transactionReceiptProcessor);
```

If you require further information, the FastRawTransactionManagerIT demonstrates the polling and queuing approaches.

5.5.12 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(
             <param1>,
             ...).send();
```

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 Flowable filter instead which will listen for events associated with the smart contract:

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

For more information on working with Flowable 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.

5.5.13 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:

```
Type result = contract.someMethod(<param1>, ...).send();
```

5.5.14 Dynamic gas price and limit

When working with smart contracts you may want to specify different gas price and limit values depending on the function being invoked. You can do that by creating your own ContractGasProvider for the smart contract wrapper.

Every generated wrapper contains all smart contract method names listed as a constants, which facilitates compilation-time matching via a *switch* statement.

For example, using the Greeter contract:

```
Greeter greeter = new Greeter(...);
greeter.setGasProvider(new DefaultGasProvider() {
    @Override
    public BigInteger getGasPrice(String contractFunc) {
        switch (contractFunc) {
            case Greeter.FUNC_GREET: return BigInteger.valueOf(22_000_000_000L);
            case Greeter.FUNC_KILL: return BigInteger.valueOf(44_000_000_000L);
            default: throw new NotImplementedException();
        }
    }

    @Override
    public BigInteger getGasLimit(String contractFunc) {
        switch (contractFunc) {
            case Greeter.FUNC_GREET: return BigInteger.valueOf(4_300_000);
            case Greeter.FUNC_KILL: return BigInteger.valueOf(5_300_000);
            default: throw new NotImplementedException();
        }
    }
});
```

5.5.15 Examples

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

5.6 Application Binary Interface

The Application Binary Interface (ABI) is a data encoding scheme used in Ethereum for working with smart contracts. The types defined in the ABI are the same as those you encounter when writing Smart Contracts with Solidity - i.e. *uint8, ..., uint256, int8, ..., int256, bool, string,* etc.

The ABI module in web3j provides full support for the ABI specification, and includes:

- Java implementations of all ABI types, including conversion from and to native Java types
- Function and event support
- Plenty of unit tests

5.6.1 Type mappings

The native Java to ABI type mappings used within web3j are as follows:

- boolean -> bool
- BigInteger -> uint/int
- byte[] -> bytes
- String -> string and address types
- List<> -> dynamic/static array

BigInteger types have to be used for numeric types, as numeric types in Ethereum are 256 bit integer values.

Fixed point types have been defined for Ethereum, but are not currently implemented in Solidity, hence web3j does not currently support them (they were provided in versions prior to 3.x). Once available in Solidity, they will be reintroduced back into the web3j ABI module.

For more information on using ABI types in Java, refer to *Solidity smart contract wrappers*.

5.6.2 Further details

Please refer to the various ABI unit tests for encoding/decoding examples.

A full ABI specification is maintained with the Solidity documentation.

5.6.3 Dependencies

This is a very lightweight module, with the only third-party dependency being Bouncy Castle for cryptographic hashing (Spongy Castle on Android). The hope is that other projects wishing to work with Ethereum's ABI on the JVM or Android will choose to make use of this module rather then write their own implementations.

5.7 Recursive Length Prefix

The Recursive Length Prefix (RLP) encoding scheme is a space efficient object serialization scheme used in Ethereum.

The specification itself is defined in the Yellow Paper, and the following page on the Ethereum Wiki.

5.7.1 RLP Types

The RLP encoder defined two supported types:

- string
- list

The list type can be nested an arbitrary number of times allowing complex data structures to be encoded.

The RLP module in web3j provides RLP encoding capabilities, with the RlpEncoderTest demonstrating encoding of a number of different values.

5.7.2 Transaction encoding

Within web3j, RLP encoding is used to encode Ethereum transaction objects into a byte array which is signed before submission to the network. The transaction types and signing logic are located within the Crypto module, with the TransactionEncoderTest providing examples of transaction signing and encoding.

5.7.3 Dependencies

This is a very lightweight module, with no other dependencies. The hope is that other projects wishing to work with Ethereum's RLP encoding on the JVM or Android will choose to make use of this module rather then write their own implementations.

5.8 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 Flowables 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.

5.8.1 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.blockFlowable(false).subscribe(block -> {
    ...
});
```

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

```
Subscription subscription = web3j.transactionFlowable().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.pendingTransactionFlowable().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.

5.8.2 Replay filters

web3j also provides filters for replaying block and transaction history.

To replay a range of blocks from the blockchain:

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

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

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

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

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

As above, but with transactions contained within blocks:

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

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

5.8.3 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.ethLogFlowable(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.

5.8.4 A note on functional composition

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

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

```
public Flowable<EthBlock> blockFlowable(
        boolean fullTransactionObjects, long pollingInterval) {
    return this.ethBlockHashFlowable(pollingInterval)
            .flatMap(blockHash ->
                        web3j.ethGetBlockByHash(blockHash, fullTransactionObjects).flowable());
}
```

Here we first create a flowable 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 flowable.

5.8.5 Further examples

Please refer to the integration test FlowableIT for further examples.

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

5.9 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, or for Arch linux users via the AUR.

```
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-3.0.0/lib/
  inflating: web3j-3.0.0/lib/core-1.0.2-all.jar
   creating: web3j-3.0.0/bin/
  inflating: web3j-3.0.0/bin/web3j
  inflating: web3j-3.0.0/bin/web3j.bat
$ ./web3j-<version>/bin/web3j

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

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

5.9.1 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> 0x<address>|<ensName>
```

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> 0x<address>|<ensName>

              _      _____ _     _
             | |    |____ (_)   (_)
__      _____| |__      / /_     _   ___
\ \ /\ / / _ \ '_ \     \ \ |   | | / _ \
 \ 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 0x9c98e381edc5
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
```

5.9.2 Solidity smart contract wrapper generator

Please refer to *Solidity smart contract wrappers*.

5.10 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 and 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. Those methods that are common to both Geth and Parity reside in the Admin module of web3j.

You can initialise a new web3j connector that supports this module using the factory method:

```
Admin web3j = Admin.build(new HttpService());  // defaults to http://localhost:8545/
PersonalUnlockAccount personalUnlockAccount = admin.personalUnlockAccount("0x000...", "a password").s
if (personalUnlockAccount.accountUnlocked()) {
    // send a transaction
}
```

For Geth specific methods, you can use the Geth connector, and for Parity you can use the associated Parity connector. The *Parity* connector also provides support for Parity's Trace module. These connectors are available in the web3j *geth* and *parity* modules respectively.

You can refer to the integration test ParityIT for further examples of working with these APIs.

5.11 Using Infura with web3j

5.11.1 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 (Rinkeby): https://rinkeby.infura.io/<your-token>

Test Ethereum Network (Kovan): https://kovan.infura.io/<your-token>

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

For obtaining ether to use in these networks, you can refer to *Ethereum testnets*

5.11.2 InfuraHttpClient

The web3j infura module provides an Infura HTTP client (InfuraHttpService) which provides support for the Infura specific *Infura-Ethereum-Preferred-Client* header. This allows you to specify whether you want a Geth or Parity client to respond to your request. You can create the client just like the regular HTTPClient:

```
Web3j web3 = Web3j.build(new HttpService("https://rinkeby.infura.io/<your-token>"));
Web3ClientVersion web3ClientVersion = web3.web3ClientVersion().send();
System.out.println(web3ClientVersion.getWeb3ClientVersion());
```

```
Geth/v1.7.2-stable-1db4ecdc/darwin-amd64/go1.9.1
```

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.

For further information, refer to the Infura docs.

5.11.3 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.

5.12 Ethereum Name Service

The Ethereum Name Service (ENS) provides a human readable names to identify addresses on the Ethereum network. It is similar to the internet's domain name service (DNS) which provides human-readable domain names which are mapped to IP addresses.

In the case of ENS, the addresses are either wallet or smart contract addresses.

E.g. instead of using the wallet address *0x19e03255f667bdfd50a32722df860b1eeaf4d635*, you can use *web3j.eth*.

5.12.1 Usage in web3j

You can use ENS names anywhere you wish to transact in web3j. In practice this means, in smart contract wrappers, when you load them, such as:

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

Also, when performing Ether transfers, such as using the command line tools:

```
$ web3j wallet send <walletfile> 0x<address>|<ensName>
```

5.12.2 web3j implementation

Behind the scenes, whenever you using web3j's transaction managers (which are derived from the ManagedTransaction class), the EnsResolver is invoked to perform an ENS lookup if applicable.

The resolution process is as follows:

- Check to see if our Ethereum node is fully synced
- If not fail
- **If it is synced, check the timestamp on the most recent block it has.**
    - If it's more than 3 minutes old, fail.
    - Otherwise perform the lookup

If you need to change the threshold parameter of what constitutes being synced to something other then 3 minutes, this can be done via the *setSyncThreshold* method in the ManagedTransaction class.

5.12.3 Unicode Technical Standard (UTS) #46

UTS #46 is the standard used to sanitise input on domain names. The web3j ENS implementation peforms this santisation on all inputs before attempting resolution. For details of the implementation, refer to the NameHash class.

5.12.4 Registering domain names

Currently, web3j only supports the resolution of ENS domains. It does not support the registration. For instructions on how to do this, refer to the ENS quickstart.

5.13 Contracts supported by web3j

5.13.1 EIP20

ERC20 tokens are supported via ERC20 contract wrapper as defined in EIP20 To fetch your token balance you can simply do:

```
ERC20 contract = ERC20.load(tokenAddress, web3j, txManager, gasPriceProvider);
BigInteger balance = contract.balanceOf(account).send();
```

5.13.2 EIP165

Smart contract interfaces support and discovery as defined in EIP165 To check whether token contract supports particular interface:

```
ERC165 contract = ERC165.load(tokenAddress, web3j, txManager, gasPriceProvider);
Boolean isSupported = contract.supportsInterface(interfaceID).send();
```

5.13.3 EIP721

Support for non-fungible tokens, also known as deeds as defined in EIP721 This contains the following contract wrappers:

- ERC721 is a set of methods that NFT should support
- ERC721Metadata optional metadata extension for NFT
- ERC721Enumerable optional enumeration extension for NFT

5.14 Troubleshooting

5.14.1 Do you have a sample web3j project

Yes, refer to the web3j sample project outlined in the Quickstart.

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

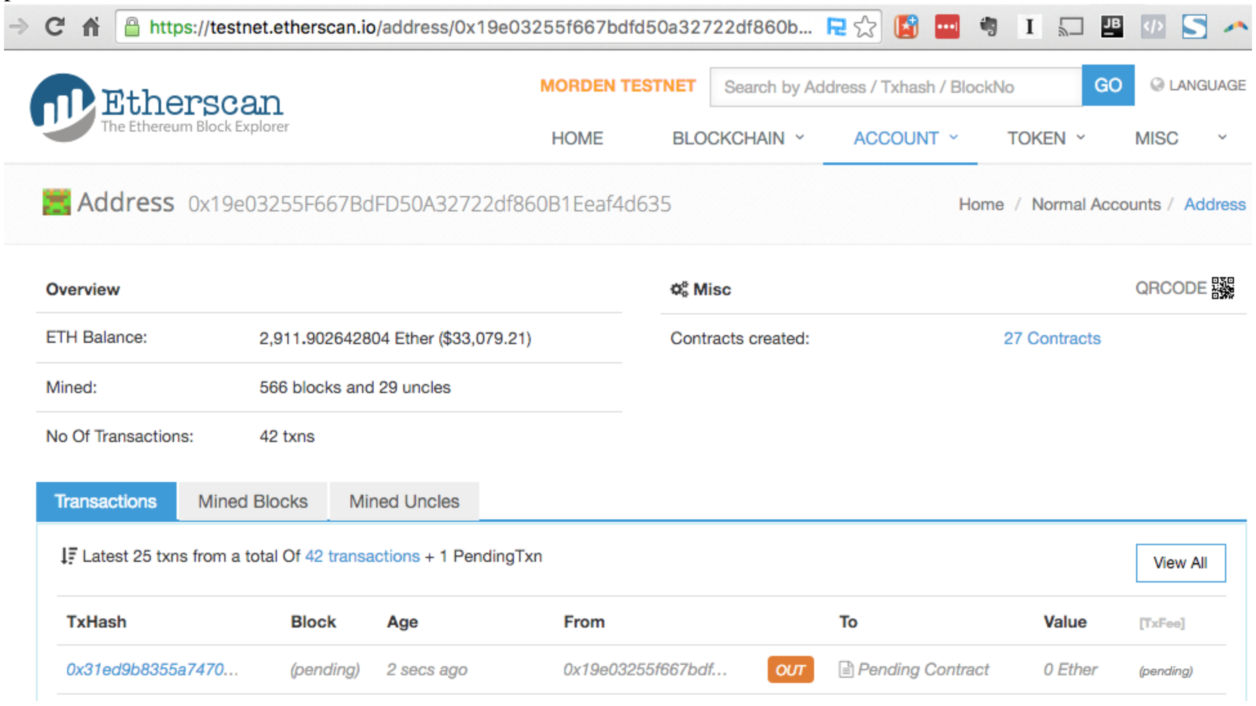
After creating and sending a transaction, you receive a transaction hash, however calling eth_getTransactionReceipt always returns a blank value, indicating the transaction has not been mined:

```
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(0xeaac9aab7f9aeab189acd8714c5a60c7424f86820884b815c4448cfcc
```

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.


https://testnet.etherscan.io/address/0x19e03255f667bdfd50a32722df860b...
Etherscan
The Ethereum Block Explorer
MORDEN TESTNET
Search by Address / Txhash / BlockNo
GO
LANGUAGE
HOME
BLOCKCHAIN
ACCOUNT
TOKEN
MISC
Home / Normal Accounts / Address
QRCODE
Transactions
Mined Blocks
Mined Uncles
View All

Address 0x19e03255F667BdFD50A32722df860B1Eeaf4d635

Overview

ETH Balance:	2,911.902642804 Ether ($33,079.21)
Mined:	566 blocks and 29 uncles
No Of Transactions:	42 txns

Misc

Contracts created:	27 Contracts

Latest 25 txns from a total Of 42 transactions + 1 PendingTxn

TxHash	Block	Age	From		To	Value	[TxFee]
0x31ed9b8355a7470...	(pending)	2 secs ago	0x19e03255f667bdf...	OUT	Pending Contract	0 Ether	(pending)

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.

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

web3j uses the SLF4J logging facade, which you can easily integrate with your preferred logging framework. One lightweight approach is to use LOGBack, which is already configured in the integration-tests module.

Include the LOGBack dependencies listed in integration-tests/build.gradle and associated log configuration as per integration-tests/src/test/resources/logback-test.xml.

Note: if you are configuring logging for an application (not tests), you will need to ensure that the Logback dependencies are configured as *compile* dependencies, and that the configuration file is named and located in *src/main/resources/logback.xml*.

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

Please refer to the *Ethereum testnets* for how to obtain some Ether.

5.14.5 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.

5.14.6 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 =
        web3j.ethSendRawTransaction(hexValue).send();
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.

5.14.7 I've generated my smart contract wrapper, but the binary for the smart contract is empty?

If you have defined an interface in Solidity, but one of your method implementations doesn't match the original interface definitions, the produced binary will be blank.

In the following example:

```
contract Web3jToken is ERC20Basic, Ownable {
    ...
    function transfer(address _from, address _to, uint256 _value) onlyOwner returns (bool) {
    ...
}
```

We forgot to define the *from* parameter in one of the inherited contracts:

```
contract ERC20Basic {
    ...
    function transfer(address to, uint256 value) returns (bool);
    ...
}
```

The Solidity compiler will not complain about this, however, the produced binary file for the Web3jToken will be blank.

5.14.8 My ENS lookups are failing

Are you sure that you are connecting to the correct network to perform the lookup?

If web3j is telling you that the node is not in sync, you may need to change the *syncThreshold* in the *ENS resolver*.

5.14.9 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

5.14.10 Where can I get commercial support for web3j?

Commercial support and training is available from blk.io.

5.15 Projects using web3j

- Ether Wallet by @vikulin
- eth-contract-api by @adridadou
- Ethereum Paper Wallet by @matthiaszimmermann
- web3j-scala by @mslinn

5.16 Companies using web3j

- Amberdata
- blk.io
- comitFS
- ConsenSys
- ING
- Othera
- TrustWallet

5.17 Developer Guide

5.17.1 Dependency management

We recommend you use formal releases of web3j, these can be found on most public maven repositories.

Release versions follow the `<major>.<minor>.<build>` convention, for example: 4.2.0

Snapshot versions of web3j follow the `<major>.<minor>.<build>-SNAPSHOT` convention, for example: 4.2.0-SNAPSHOT.

If you would like to use snapshots instead please add a new maven repository pointing to:

```
https://oss.sonatype.org/content/repositories/snapshots
```

Please refer to the maven or gradle documentation for further detail.

Sample gradle configuration:

```
repositories {
   maven {
      url "https://oss.sonatype.org/content/repositories/snapshots"
   }
}
```

Sample maven configuration:

```
<repositories>
  <repository>
    <id>sonatype-snasphots</id>
    <name>Sonatype snapshots repo</name>
    <url>https://oss.sonatype.org/content/repositories/snapshots</url>
  </repository>
</repositories>
```

5.17.2 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 (excluding integration tests):

```
$ ./gradlew check
```

To run the integration tests:

```
$ ./gradlew  -Pintegration-tests=true :integration-tests:test
```

5.17.3 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
```

5.18 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 dedicated Parity dashboard, Rinkeby testnet dashboard, and one for the Kovan testnet.
- Ethereum reddit

5.19 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