---

title: JSON-RPC API description: A stateless, light-weight remote procedure call (RPC) protocol for Ethereum clients. lang: en

---

In order for a software application to interact with the Ethereum blockchain - either by reading blockchain data or sending transactions to the network - it must connect to an Ethereum node.

For this purpose, every Ethereum client implements a JSON-RPC specification, so there is a uniform set of methods that applications can rely on regardless of the specific node or client implementation.

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. It defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over HTTP, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

Client implementations {#client-implementations}

Ethereum clients each may utilize different programming languages when implementing the JSON-RPC specification. See individual client documentation for further details related to specific programming languages. We recommend checking the documentation of each client for the latest API support information.

Convenience Libraries {#convenience-libraries}

While you may choose to interact directly with Ethereum clients via the JSON-RPC API, there are often easier options for dapp developers. Many JavaScript and backend API libraries exist to provide wrappers on top of the JSON-RPC API. With these libraries, developers can write intuitive, one-line methods in the programming language of their choice to initialize JSON-RPC requests (under the hood) that interact with Ethereum.

Consensus client APIs {#consensus-clients}

This page deals mainly with the JSON-RPC API used by Ethereum execution clients. However, consensus clients also have an RPC API that allows users to query information about the node, request Beacon blocks, Beacon state, and other consensus-related information directly from a node. This API is documented on the Beacon API webpage.

An internal API is also used for inter-client communication within a node - that is, it enables the consensus client and execution client to swap data. This is called the 'Engine API' and the specs are available on GitHub.

Execution client spec {#spec}

Read the full JSON-RPC API spec on GitHub

Conventions {#conventions}

Hex value encoding {#hex-encoding}

Two key data types get passed over JSON: unformatted byte arrays and quantities. Both are passed with a hex encoding but with different requirements for formatting.

Quantities {#quantities-encoding}

When encoding quantities (integers, numbers): encode as hex, prefix with "0x", the most compact representation (slight exception: zero should be represented as "0x0").

Here are some examples:

- 0x41 (65 in decimal)
- 0x400 (1024 in decimal)
- WRONG: 0x (should always have at least one digit - zero is "0x0")
- WRONG: 0x0400 (no leading zeroes allowed)
- WRONG: ff (must be prefixed 0x)

Unformatted data {#unformatted-data-encoding}

When encoding unformatted data (byte arrays, account addresses, hashes, bytecode arrays): encode as hex, prefix with "0x", two hex digits per byte.

Here are some examples:

- 0x41 (size 1, "A")
- 0x004200 (size 3, "\0B\0")
- 0x (size 0, "")
- WRONG: 0xf0f0f (must be even number of digits)
- WRONG: 004200 (must be prefixed 0x)

The default block parameter {#default-block}

The following methods have an extra default block parameter:

- eth_getBalance
- eth_getCode
- eth_getTransactionCount
- eth_getStorageAt
- eth_call

When requests are made that act on the state of Ethereum, the last default block parameter determines the height of the block.

The following options are possible for the defaultBlock parameter:

- `HEX String` - an integer block number
- `String "earliest"` for the earliest/genesis block
- `String "latest"` - for the latest mined block
- `String "safe"` - for the latest safe head block
- `String "finalized"` - for the latest finalized block
- `String "pending"` - for the pending state/transactions

Examples

On this page we provide examples of how to use individual JSON_RPC API endpoints using the command line tool curl. These individual endpoint examples are found below in the Curl examples section. Further down the page, we also provide an end-to-end example for compiling and deploying a smart contract using a Geth node, the JSON_RPC API and curl.

Curl examples {#curl-examples}

Examples of using the JSON_RPC API by makingcurl requests to an Ethereum node are provided below. Each example includes a description of the specific endpoint, its parameters, return type, and a worked example of how it should be used.

The curl requests might return an error message relating to the content type. This is because the `-data` option sets the content type to`application/x-www-form-urlencoded`. If your node does complain about this, manually set the header by placing `-H "Content-Type: application/json"` at the start of the call. The examples also do not include the URL/IP & port combination which must be the last argument given to curl (e.g. `127.0.0.1:8545`). A complete curl request including these additional data takes the following form:

```
shell curl -H "Content-Type: application/json" -X POST --data '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":67}' 127.0.0.1:8545
```

Gossip, State, History {#gossip-state-history}

A handful of core JSON-RPC methods require data from the Ethereum network, and fall neatly into three main categories*Gossip, State, and History*. Use the links in these sections to jump to each method, or use the table of contents to explore the whole list of methods.

Gossip Methods {#gossip-methods}

These methods track the head of the chain. This is how transactions make their way around the network, find their way into blocks, and how clients find out about new blocks.

- eth_blockNumber
- eth_sendRawTransaction

State Methods {#state_methods}

Methods that report the current state of all the data stored. The "state" is like one big shared piece of RAM, and includes account balances, contract data, and gas estimations.

- eth_getBalance
- eth_getStorageAt
- eth_getTransactionCount
- eth_getCode
- eth_call
- eth_estimateGas

History Methods {#history_methods}

Fetches historical records of every block back to genesis. This is like one large append-only file, and includes all block headers, block bodies, uncle blocks, and transaction receipts.

- eth_getBlockTransactionCountByHash
- eth_getBlockTransactionCountByNumber
- eth_getUncleCountByBlockHash
- eth_getUncleCountByBlockNumber
- eth_getBlockByHash
- eth_getBlockByNumber
- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex
- eth_getTransactionByBlockNumberAndIndex
- eth_getTransactionReceipt
- eth_getUncleByBlockHashAndIndex
- eth_getUncleByBlockNumberAndIndex

JSON-RPC API Methods {#json-rpc-methods}

web3_clientVersion {#web3_clientversion}

Returns the current client version.

Parameters

None

Returns

`String` - The current client version

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":67}' // Result { "id":67, "jsonrpc":"2.0", "result": "Geth/v1.12.1-stable/linux-amd64/go1.19.1" }
```

web3_sha3 {#web3_sha3}

Returns Keccak-256 (*not* the standardized SHA3-256) of the given data.

Parameters

1. `DATA` - the data to convert into a SHA3 hash

```
js params: ["0x68656c6c6f20776f726c64"]
```

Returns

`DATA` - The SHA3 result of the given string.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"web3_sha3","params":["0x68656c6c6f20776f726c64"],"id":64}' // Result { "id":64, "jsonrpc": "2.0", "result": "0x47173285a8d7341e5e972fc677286384f802f8ef42a5ec5f03bbfa254cb01fad" }
```

net_version {#net_version}

Returns the current network id.

Parameters

None

Returns

`String` - The current network id.

The full list of current network IDs is available at chainlist.org. Some common ones are:

- `1`: Ethereum Mainnet
- `5`: Goerli testnet
- `11155111`: Sepolia testnet

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"net_version","params":[],"id":67}' // Result { "id":67, "jsonrpc": "2.0", "result": "3" }
```

net_listening {#net_listening}

Returns `true` if client is actively listening for network connections.

Parameters

None

Returns

`Boolean` - `true` when listening, otherwise `false`.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"net_listening","params":[],"id":67}' // Result { "id":67, "jsonrpc":"2.0", "result":true }
```

net_peerCount {#net_peercount}

Returns number of peers currently connected to the client.

Parameters

None

Returns

`QUANTITY` - integer of the number of connected peers.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"net_peerCount","params":[],"id":74}' // Result { "id":74, "jsonrpc": "2.0", "result": "0x2" // 2 }
```

eth_protocolVersion {#eth_protocolversion}

Returns the current Ethereum protocol version. Note that this method is not available in Geth.

Parameters

None

Returns

`String` - The current Ethereum protocol version

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_protocolVersion","params":[],"id":67}' // Result { "id":67, "jsonrpc": "2.0", "result": "54" }
```

eth_syncing {#eth_syncing}

Returns an object with data about the sync status or `false`.

Parameters

None

Returns

The precise return data varies between client implementations. All clients return `False` when the node is not syncing, and all clients return the following fields.

`Object|Boolean`, An object with sync status data or `FALSE`, when not syncing:

- `startingBlock`: `QUANTITY` - The block at which the import started (will only be reset, after the sync reached his head)
- `currentBlock`: `QUANTITY` - The current block, same as eth_blockNumber
- `highestBlock`: `QUANTITY` - The estimated highest block

However, the individual clients may also provide additional data. For example Geth returns the following:

```
json { "jsonrpc": "2.0", "id": 1, "result": { "currentBlock": "0x3cf522", "healedBytecodeBytes": "0x0", "healedBytecodes": "0x0", "healedTrienodes": "0x0", "healingBytecode": "0x0", "healingTrienodes": "0x0", "highestBlock": "0x3e0e41", "startingBlock": "0x3cbed5", "syncedAccountBytes": "0x0", "syncedAccounts": "0x0", "syncedBytecodeBytes": "0x0", "syncedBytecodes": "0x0", "syncedStorage": "0x0", "syncedStorageBytes": "0x0" } }
```

Whereas Besu returns:

```
json { "jsonrpc": "2.0", "id": 51, "result": { "startingBlock": "0x0", "currentBlock": "0x1518", "highestBlock": "0x9567a3", "pulledStates": "0x203ca", "knownStates": "0x200636" } }
```

Refer to the documentation for your specific client for more details.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_syncing","params":[],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result": { startingBlock: '0x384', currentBlock: '0x386', highestBlock: '0x454' } } // Or when not syncing { "id":1, "jsonrpc": "2.0", "result": false }
```

eth_coinbase {#eth_coinbase}

Returns the client coinbase address.

Parameters

None

Returns

`DATA`, 20 bytes - the current coinbase address.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_coinbase","params":[],"id":64}' // Result { "id":64, "jsonrpc": "2.0", "result":
"0x407d73d8a49eeb85d32cf465507dd71d507100c1" }
```

eth_chainId {#eth_chainId}

Returns the chain ID used for signing replay-protected transactions.

Parameters

None

Returns

`chainId`, hexadecimal value as a string representing the integer of the current chain id.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_chainId","params":[],"id":67}' // Result { "id":67, "jsonrpc": "2.0", "result": "0x1" }
```

eth_mining {#eth_mining}

Returns `true` if client is actively mining new blocks. This can only return`true` for proof-of-work networks and may not be available in some clients sinceThe Merge.

Parameters

None

Returns

`Boolean` - returns `true` of the client is mining, otherwise`false`.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_mining","params":[],"id":71}' // { "id":71, "jsonrpc": "2.0", "result": true }
```

eth_hashrate {#eth_hashrate}

Returns the number of hashes per second that the node is mining with. This can only return`true` for proof-of-work networks and may not be available in some clients sinceThe Merge.

Parameters

None

Returns

`QUANTITY` - number of hashes per second.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_hashrate","params":[],"id":71}' // Result { "id":71, "jsonrpc": "2.0", "result": "0x38a" }
```

eth_gasPrice {#eth_gasprice}

Returns an estimate of the current price per gas in wei. For example, the Besu client examines the last 100 blocks and returns the median gas unit price by default.

Parameters

None

Returns

`QUANTITY` - integer of the current gas price in wei.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":73}' // Result { "id":73, "jsonrpc": "2.0", "result": "0x1dfd14000" // 8049999872 Wei }
```

eth_accounts {#eth_accounts}

Returns a list of addresses owned by client.

Parameters

None

Returns

`Array of DATA`, 20 Bytes - addresses owned by the client.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_accounts","params":[],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result":
["0x407d73d8a49eeb85d32cf465507dd71d507100c1"] }
```

eth_blockNumber {#eth_blocknumber}

Returns the number of most recent block.

Parameters

None

Returns

`QUANTITY` - integer of the current block number the client is on.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":83}' // Result { "id":83, "jsonrpc": "2.0", "result": "0x4b7" // 1207 }
```

eth_getBalance {#eth_getbalance}

Returns the balance of the account of given address.

Parameters

1. `DATA`, 20 Bytes - address to check for balance.
2. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
js params: ["0x407d73d8a49eeb85d32cf465507dd71d507100c1", "latest"]
```

Returns

`QUANTITY` - integer of the current balance in wei.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBalance","params":["0x407d73d8a49eeb85d32cf465507dd71d507100c1", "latest"],"id":1}' // Result { "id":1,
"jsonrpc": "2.0", "result": "0x0234c8a3397aab58" // 158972490234375000 }
```

eth_getStorageAt {#eth_getstorageat}

Returns the value from a storage position at a given address.

Parameters

1. `DATA`, 20 Bytes - address of the storage.
2. `QUANTITY` - integer of the position in the storage.
3. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

Returns

`DATA` - the value at this storage position.

Example Calculating the correct position depends on the storage to retrieve. Consider the following contract deployed at `0x295a70b2de5e3953354a6a8344e616ed314d7251` by address `0x391694e7e0b0cce554cb130d723a9d27458f9298`.

```
contract Storage { uint pos0; mapping(address => uint) pos1; function Storage() { pos0 = 1234; pos1[msg.sender] = 5678; } }
```

Retrieving the value of pos0 is straight forward:

```
js curl -X POST --data '{"jsonrpc":"2.0", "method": "eth_getStorageAt", "params": ["0x295a70b2de5e3953354a6a8344e616ed314d7251", "0x0", "latest"], "id": 1}' localhost:8545
{"jsonrpc":"2.0","id":1,"result":"0x00000000000000000000000000000000000000000000000000000000000004d2"}
```

Retrieving an element of the map is harder. The position of an element in the map is calculated with:

```
js keccack(LeftPad32(key, 0), LeftPad32(map position, 0))
```

This means to retrieve the storage on pos1["0x391694e7e0b0cce554cb130d723a9d27458f9298"] we need to calculate the position with:

```
js keccak( decodeHex( "000000000000000000000000391694e7e0b0cce554cb130d723a9d27458f9298" + "0000000000000000000000000000000000000000000000000000000000000001" ) )
```

The geth console which comes with the web3 library can be used to make the calculation:

```js

    var key = "000000000000000000000000391694e7e0b0cce554cb130d723a9d27458f9298" + "0000000000000000000000000000000000000000000000000000000000000001" undefined
    web3.sha3(key, {"encoding": "hex"}) "0x6661e9d6d8b923d5bbaab1b96e1dd51ff6ea2a93520fdc9eb75d059238b8c5e9" ```
```

Now to fetch the storage:

```
js curl -X POST --data '{"jsonrpc":"2.0", "method": "eth_getStorageAt", "params": ["0x295a70b2de5e3953354a6a8344e616ed314d7251",
"0x6661e9d6d8b923d5bbaab1b96e1dd51ff6ea2a93520fdc9eb75d059238b8c5e9", "latest"], "id": 1}' localhost:8545
{"jsonrpc":"2.0","id":1,"result":"0x000000000000000000000000000000000000000000000000000000000000162e"}
```

eth_getTransactionCount {#eth_gettransactioncount}

Returns the number of transactions *sent* from an address.

Parameters

1. `DATA`, 20 Bytes - address.
2. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
js params: [ "0x407d73d8a49eeb85d32cf465507dd71d507100c1", "latest", // state at the latest block ]
```

Returns

`QUANTITY` - integer of the number of transactions send from this address.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionCount","params":["0x407d73d8a49eeb85d32cf465507dd71d507100c1","latest"],"id":1}' // Result { "id":1,
"jsonrpc": "2.0", "result": "0x1" // 1 }
```

eth_getBlockTransactionCountByHash {#eth_getblocktransactioncountbyhash}

Returns the number of transactions in a block from a block matching the given block hash.

Parameters

1. `DATA`, 32 Bytes - hash of a block

```
js params: ["0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"]
```

Returns

`QUANTITY` - integer of the number of transactions in this block.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByHash","params":
["0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result": "0xb" // 11 }
```

eth_getBlockTransactionCountByNumber {#eth_getblocktransactioncountbynumber}

Returns the number of transactions in a block matching the given block number.

Parameters

1. `QUANTITY|TAG` - integer of a block number, or the string `"earliest"`, `"latest"` or `"pending"`, as in the default block parameter.

```
js params: [ "0xe8", // 232 ]
```

Returns

`QUANTITY` - integer of the number of transactions in this block.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByNumber","params":["0xe8"],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result": "0xa"
// 10 }
```

eth_getUncleCountByBlockHash {#eth_getunclecountbyblockhash}

Returns the number of uncles in a block from a block matching the given block hash.

Parameters

1. `DATA`, 32 Bytes - hash of a block

```
js params: ["0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"]
```

Returns

`QUANTITY` - integer of the number of uncles in this block.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleCountByBlockHash","params":["0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"],"id":1}'
// Result { "id":1, "jsonrpc": "2.0", "result": "0x1" // 1 }
```

eth_getUncleCountByBlockNumber {#eth_getunclecountbyblocknumber}

Returns the number of uncles in a block from a block matching the given block number.

Parameters

1. `QUANTITY|TAG` - integer of a block number, or the string "latest", "earliest" or "pending", see thedefault block parameter

```
js params: [ "0xe8", // 232 ]
```

Returns

`QUANTITY` - integer of the number of uncles in this block.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleCountByBlockNumber","params":["0xe8"],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result": "0x1" // 1 }
```

eth_getCode {#eth_getcode}

Returns code at a given address.

Parameters

1. `DATA`, 20 Bytes - address
2. `QUANTITY|TAG` - integer block number, or the string`"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
js params: [ "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b", "0x2", // 2 ]
```

Returns

`DATA` - the code from the given address.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getCode","params":["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b", "0x2"],"id":1}' // Result { "id":1, "jsonrpc":
"2.0", "result": "0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f25b600060078202905091905056" }
```

eth_sign {#eth_sign}

The sign method calculates an Ethereum specific signature with:`sign(keccak256("\x19Ethereum Signed Message:\n" + len(message) + message))`.

By adding a prefix to the message makes the calculated signature recognizable as an Ethereum specific signature. This prevents misuse where a malicious dapp can sign arbitrary data (e.g. transaction) and use the signature to impersonate the victim.

Note: the address to sign with must be unlocked.

Parameters

1. `DATA`, 20 Bytes - address
2. `DATA`, N Bytes - message to sign

Returns

`DATA`: Signature

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sign","params":["0x9b2055d370f73ec7d8a03e965129118dc8f5bf83", "0xdeadbeaf"],"id":1}' // Result { "id":1, "jsonrpc":
"2.0", "result": "0xa3f20717a250c2b0b729b7e5becbff67fdaef7e0699da4de7ca5895b02a170a12d887fd3b17bfdce3481f10bea41f45ba9f709d39ce8325427b57afcfc994cee1b" }
```

eth_signTransaction {#eth_signtransaction}

Signs a transaction that can be submitted to the network at a later time using witheth_sendRawTransaction.

Parameters

1. `Object` - The transaction object
2. `type`:
3. `from`: `DATA`, 20 Bytes - The address the transaction is sent from.
4. `to`: `DATA`, 20 Bytes - (optional when creating new contract) The address the transaction is directed to.
5. `gas`: `QUANTITY` - (optional, default: 90000) Integer of the gas provided for the transaction execution. It will return unused gas.
6. `gasPrice`: `QUANTITY` - (optional, default: To-Be-Determined) Integer of the gasPrice used for each paid gas, in Wei.
7. `value`: `QUANTITY` - (optional) Integer of the value sent with this transaction, in Wei.

8. `data`: `DATA` - The compiled code of a contract OR the hash of the invoked method signature and encoded parameters.
9. `nonce`: `QUANTITY` - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

Returns

`DATA`, The RLP-encoded transaction object signed by the specified account.

Example

```
js // Request curl -X POST --data '{"id": 1,"jsonrpc": "2.0","method": "eth_signTransaction","params":
[{"data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675","from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155","gas": "0x76c0","gasPrice":
"0x9184e72a000","to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567","value": "0x9184e72a"}]}' // Result { "id": 1, "jsonrpc": "2.0", "result":
"0xa3f20717a250c2b0b729b7e5becbff67fdaef7e0699da4de7ca5895b02a170a12d887fd3b17bfdce3481f10bea41f45ba9f709d39ce8325427b57afcfc994cee1b" }
```

eth_sendTransaction {#eth_sendtransaction}

Creates new message call transaction or a contract creation, if the data field contains code, and signs it using the account specified in `from`.

Parameters

1. `Object` - The transaction object

2. `from`: `DATA`, 20 Bytes - The address the transaction is sent from.

3. `to`: `DATA`, 20 Bytes - (optional when creating new contract) The address the transaction is directed to.
4. `gas`: `QUANTITY` - (optional, default: 90000) Integer of the gas provided for the transaction execution. It will return unused gas.
5. `gasPrice`: `QUANTITY` - (optional, default: To-Be-Determined) Integer of the gasPrice used for each paid gas.
6. `value`: `QUANTITY` - (optional) Integer of the value sent with this transaction.
7. `input`: `DATA` - The compiled code of a contract OR the hash of the invoked method signature and encoded parameters.
8. `nonce`: `QUANTITY` - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

```
js params: [ { from: "0xb60e8dd61c5d32be8058bb8eb970870f07233155", to: "0xd46e8dd67c5d32be8058bb8eb970870f07244567", gas: "0x76c0", // 30400 gasPrice: "0x9184e72a000", //
10000000000000 value: "0x9184e72a", // 2441406250 input: "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675", }, ]
```

Returns

`DATA`, 32 Bytes - the transaction hash, or the zero hash if the transaction is not yet available.

Use eth_getTransactionReceipt to get the contract address, after the transaction was mined, when you created a contract.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{see above}],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result":
"0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331" }
```

eth_sendRawTransaction {#eth_sendrawtransaction}

Creates new message call transaction or a contract creation for signed transactions.

Parameters

1. `DATA`, The signed transaction data.

```
js params: [ "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675", ]
```

Returns

`DATA`, 32 Bytes - the transaction hash, or the zero hash if the transaction is not yet available.

Use eth_getTransactionReceipt to get the contract address, after the transaction was mined, when you created a contract.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sendRawTransaction","params":[{see above}],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result":
"0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331" }
```

eth_call {#eth_call}

Executes a new message call immediately without creating a transaction on the block chain. Often used for executing read-only smart contract functions, for example the `balanceOf` for an ERC-20 contract.

Parameters

1. `Object` - The transaction call object

2. `from`: `DATA`, 20 Bytes - (optional) The address the transaction is sent from.

3. `to`: `DATA`, 20 Bytes - The address the transaction is directed to.
4. `gas`: `QUANTITY` - (optional) Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
5. `gasPrice`: `QUANTITY` - (optional) Integer of the gasPrice used for each paid gas
6. `value`: `QUANTITY` - (optional) Integer of the value sent with this transaction

7. `input`: `DATA` - (optional) Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation

8. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

Returns

`DATA` - the return value of executed contract.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_call","params":[{see above}],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result": "0x" }
```

eth_estimateGas {#eth_estimategas}

Generates and returns an estimate of how much gas is necessary to allow the transaction to complete. The transaction will not be added to the blockchain. Note that the estimate may be significantly more than the amount of gas actually used by the transaction, for a variety of reasons including EVM mechanics and node performance.

Parameters

See eth_call parameters, except that all properties are optional. If no gas limit is specified geth uses the block gas limit from the pending block as an upper bound. As a result the returned estimate might not be enough to executed the call/transaction when the amount of gas is higher than the pending block gas limit.

Returns

`QUANTITY` - the amount of gas used.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_estimateGas","params":[{see above}],"id":1}' // Result { "id":1, "jsonrpc": "2.0", "result": "0x5208" // 21000 }
```

eth_getBlockByHash {#eth_getblockbyhash}

Returns information about a block by hash.

Parameters

1. `DATA`, 32 Bytes - Hash of a block.
2. `Boolean` - If `true` it returns the full transaction objects, if`false` only the hashes of the transactions.

```
js params: [ "0xdc0818cf78f21a8e70579cb46a43643f78291264dda342ae31049421c82d21ae", false, ]
```

Returns

`Object` - A block object, or`null` when no block was found:

- `number`: `QUANTITY` - the block number.`null` when its pending block.
- `hash`: `DATA`, 32 Bytes - hash of the block.`null` when its pending block.
- `parentHash`: `DATA`, 32 Bytes - hash of the parent block.
- `nonce`: `DATA`, 8 Bytes - hash of the generated proof-of-work.`null` when its pending block.
- `sha3Uncles`: `DATA`, 32 Bytes - SHA3 of the uncles data in the block.
- `logsBloom`: `DATA`, 256 Bytes - the bloom filter for the logs of the block.`null` when its pending block.
- `transactionsRoot`: `DATA`, 32 Bytes - the root of the transaction trie of the block.
- `stateRoot`: `DATA`, 32 Bytes - the root of the final state trie of the block.
- `receiptsRoot`: `DATA`, 32 Bytes - the root of the receipts trie of the block.
- `miner`: `DATA`, 20 Bytes - the address of the beneficiary to whom the mining rewards were given.
- `difficulty`: `QUANTITY` - integer of the difficulty for this block.
- `totalDifficulty`: `QUANTITY` - integer of the total difficulty of the chain until this block.
- `extraData`: `DATA` - the "extra data" field of this block.
- `size`: `QUANTITY` - integer the size of this block in bytes.
- `gasLimit`: `QUANTITY` - the maximum gas allowed in this block.
- `gasUsed`: `QUANTITY` - the total used gas by all transactions in this block.
- `timestamp`: `QUANTITY` - the unix timestamp for when the block was collated.
- `transactions`: `Array` - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- `uncles`: `Array` - Array of uncle hashes.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByHash","params":["0xdc0818cf78f21a8e70579cb46a43643f78291264dda342ae31049421c82d21ae", false],"id":1}' //
Result { { "jsonrpc": "2.0", "id": 1, "result": { "difficulty": "0x4ea3f27bc", "extraData": "0x476574682f4c5649562f76312e302e302f6c696e75782f676f312e342e32", "gasLimit": "0x1388",
"gasUsed": "0x0", "hash": "0xdc0818cf78f21a8e70579cb46a43643f78291264dda342ae31049421c82d21ae", "logsBloom":
"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
"miner": "0xbb7b8287f3f0a933474a79eae42cbca977791171", "mixHash": "0x4fffe9ae21f1c9e15207b1f472d5bbdd68c9595d461666602f2be20daf5e7843", "nonce": "0x689056015818adbe", "number":
"0x1b4", "parentHash": "0xe99e022112df268087ea7eafaf4790497fd21dbeeb6bd7a1721df161a6657a54", "receiptsRoot": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
"sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347", "size": "0x220", "stateRoot":
"0xddc8b0234c2e0cad087c8b389aa7ef01f7d79b2570bccb77ce48648aa61c904d", "timestamp": "0x55ba467c", "totalDifficulty": "0x78ed983323d", "transactions": [ ], "transactionsRoot":
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421", "uncles": [ ] } }
```

eth_getBlockByNumber {#eth_getblockbynumber}

Returns information about a block by block number.

Parameters

1. `QUANTITY|TAG` - integer of a block number, or the string`"earliest"`, `"latest"` or `"pending"`, as in the[default block parameter](#).
2. `Boolean` - If `true` it returns the full transaction objects, if`false` only the hashes of the transactions.

```
js params: [ "0x1b4", // 436 true, ]
```

Returns See [eth_getBlockByHash](#)

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["0x1b4", true],"id":1}'
```

Result see [eth_getBlockByHash](#)

eth_getTransactionByHash {#eth_gettransactionbyhash}

Returns the information about a transaction requested by transaction hash.

Parameters

1. `DATA`, 32 Bytes - hash of a transaction

```
js params: ["0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b"]
```

Returns

`Object` - A transaction object, or`null` when no transaction was found:

- `blockHash`: `DATA`, 32 Bytes - hash of the block where this transaction was in.`null` when its pending.
- `blockNumber`: `QUANTITY` - block number where this transaction was in.`null` when its pending.
- `from`: `DATA`, 20 Bytes - address of the sender.
- `gas`: `QUANTITY` - gas provided by the sender.
- `gasPrice`: `QUANTITY` - gas price provided by the sender in Wei.
- `hash`: `DATA`, 32 Bytes - hash of the transaction.
- `input`: `DATA` - the data send along with the transaction.
- `nonce`: `QUANTITY` - the number of transactions made by the sender prior to this one.
- `to`: `DATA`, 20 Bytes - address of the receiver.`null` when its a contract creation transaction.
- `transactionIndex`: `QUANTITY` - integer of the transactions index position in the block.`null` when its pending.
- `value`: `QUANTITY` - value transferred in Wei.
- `v`: `QUANTITY` - ECDSA recovery id
- `r`: `QUANTITY` - ECDSA signature r

- `s`: `QUANTITY` - ECDSA signature s

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByHash","params":["0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b"],"id":1}' //
Result { "jsonrpc":"2.0", "id":1, "result":{ "blockHash":"0x1d59ff54b1eb26b013ce3cb5fc9dab3705b415a67127a003c3e61eb445bb8df2", "blockNumber":"0x5daf3b", // 6139707
"from":"0xa7d9ddbe1f17865597fbd27ec712455208b6b76d", "gas":"0xc350", // 50000 "gasPrice":"0x4a817c800", // 20000000000
"hash":"0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b", "input":"0x68656c6c6f21", "nonce":"0x15", // 21 "to":"0xf02c1c8e6114b1dbe8937a39260b5b0a374432bb",
"transactionIndex":"0x41", // 65 "value":"0xf3dbb76162000", // 4290000000000000 "v":"0x25", // 37 "r":"0x1b5e176d927f8e9ab405058b2d2457392da3e20f328b16ddabcebc33eaac5fea",
"s":"0x4ba69724e8f69de52f0125ad8b3c5c2cef33019bac3249e2c0a2192766d1721c" } }
```

eth_getTransactionByBlockHashAndIndex {#eth_gettransactionbyblockhashandindex}

Returns information about a transaction by block hash and transaction index position.

Parameters

1. `DATA`, 32 Bytes - hash of a block.
2. `QUANTITY` - integer of the transaction index position.

```
js params: [ "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331", "0x0", // 0 ]
```

Returns See eth_getTransactionByHash

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByBlockHashAndIndex","params":["0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b",
"0x0"],"id":1}'
```

Result see eth_getTransactionByHash

eth_getTransactionByBlockNumberAndIndex {#eth_gettransactionbyblocknumberandindex}

Returns information about a transaction by block number and transaction index position.

Parameters

1. `QUANTITY|TAG` - a block number, or the string `"earliest"`, `"latest"` or `"pending"`, as in the default block parameter.
2. `QUANTITY` - the transaction index position.

```
js params: [ "0x29c", // 668 "0x0", // 0 ]
```

Returns See eth_getTransactionByHash

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByBlockNumberAndIndex","params":["0x29c", "0x0"],"id":1}'
```

Result see eth_getTransactionByHash

eth_getTransactionReceipt {#eth_gettransactionreceipt}

Returns the receipt of a transaction by transaction hash.

Note That the receipt is not available for pending transactions.

Parameters

1. `DATA`, 32 Bytes - hash of a transaction

```
js params: ["0x85d995eba9763907fdf35cd2034144dd9d53ce32cbec21349d4b12823c6860c5"]
```

Returns `Object` - A transaction receipt object, or `null` when no receipt was found:

- `transactionHash`: `DATA`, 32 Bytes - hash of the transaction.
- `transactionIndex`: `QUANTITY` - integer of the transactions index position in the block.
- `blockHash`: `DATA`, 32 Bytes - hash of the block where this transaction was in.
- `blockNumber`: `QUANTITY` - block number where this transaction was in.
- `from`: `DATA`, 20 Bytes - address of the sender.
- `to`: `DATA`, 20 Bytes - address of the receiver. null when its a contract creation transaction.
- `cumulativeGasUsed` : `QUANTITY` - The total amount of gas used when this transaction was executed in the block.
- `effectiveGasPrice` : `QUANTITY` - The sum of the base fee and tip paid per unit of gas.
- `gasUsed`: `QUANTITY` - The amount of gas used by this specific transaction alone.
- `contractAddress`: `DATA`, 20 Bytes - The contract address created, if the transaction was a contract creation, otherwise `null`.
- `logs`: `Array` - Array of log objects, which this transaction generated.
- `logsBloom`: `DATA`, 256 Bytes - Bloom filter for light clients to quickly retrieve related logs.
- `type`: `QUANTITY` - integer of the transaction type, `0x0` for legacy transactions, `0x1` for access list types, `0x2` for dynamic fees.

It also returns *either* :

- `root` : `DATA` 32 bytes of post-transaction stateroot (pre Byzantium)
- `status`: `QUANTITY` either `1` (success) or `0` (failure)

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionReceipt","params":["0x85d995eba9763907fdf35cd2034144dd9d53ce32cbec21349d4b12823c6860c5"],"id":1}' //
Result { "jsonrpc": "2.0", "id": 1, "result": { "blockHash": "0xa957d47df264a31badc3ae823e10ac1d444b098d9b73d204c40426e57f47e8c3", "blockNumber": "0xeff35f", "contractAddress":
null, // string of the address if it was created "cumulativeGasUsed": "0xa12515", "effectiveGasPrice": "0x5a9c688d4", "from": "0x6221a9c005f6e47eb398fd867784cacfdcfff4e7",
"gasUsed": "0xb4c8", "logs": [{ // logs as returned by getFilterLogs, etc. }], "logsBloom": "0x00...0", // 256 byte bloom filter "status": "0x1", "to":
"0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2", "transactionHash": "0x85d995eba9763907fdf35cd2034144dd9d53ce32cbec21349d4b12823c6860c5", "transactionIndex": "0x66", "type": "0x2" } }
```

eth_getUncleByBlockHashAndIndex {#eth_getunclebyblockhashandindex}

Returns information about a uncle of a block by hash and uncle index position.

Parameters

1. `DATA`, 32 Bytes - The hash of a block.
2. `QUANTITY` - The uncle's index position.

```
js params: [ "0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b", "0x0", // 0 ]
```

Returns See eth_getBlockByHash

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleByBlockHashAndIndex","params":["0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b",
"0x0"],"id":1}'
```

Result see eth_getBlockByHash

Note: An uncle doesn't contain individual transactions.

eth_getUncleByBlockNumberAndIndex {#eth_getunclebyblocknumberandindex}

Returns information about a uncle of a block by number and uncle index position.

Parameters

1. `QUANTITY|TAG` - a block number, or the string `"earliest"`, `"latest"` or `"pending"`, as in the default block parameter.
2. `QUANTITY` - the uncle's index position.

```
js params: [ "0x29c", // 668 "0x0", // 0 ]
```

Returns See eth_getBlockByHash

Note: An uncle doesn't contain individual transactions.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleByBlockNumberAndIndex","params":["0x29c", "0x0"],"id":1}'
```

Result see eth_getBlockByHash

eth_newFilter {#eth_newfilter}

Creates a filter object, based on filter options, to notify when the state changes (logs). To check if the state has changed, call eth_getFilterChanges.

A note on specifying topic filters: Topics are order-dependent. A transaction with a log with topics [A, B] will be matched by the following topic filters:

- `[]` "anything"
- `[A]` "A in first position (and anything after)"
- `[null, B]` "anything in first position AND B in second position (and anything after)"
- `[A, B]` "A in first position AND B in second position (and anything after)"
- `[[A, B], [A, B]]` "(A OR B) in first position AND (A OR B) in second position (and anything after)"

- **Parameters**

- `Object` - The filter options:

- `fromBlock`: `QUANTITY|TAG` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.

- `toBlock`: `QUANTITY|TAG` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
- `address`: `DATA|Array`, 20 Bytes - (optional) Contract address or a list of addresses from which logs should originate.
- `topics`: `Array of DATA`, - (optional) Array of 32 Bytes `DATA` topics. Topics are order-dependent. Each topic can also be an array of DATA with "or" options.

```
js params: [ { fromBlock: "0x1", toBlock: "0x2", address: "0x8888f1f195afa192cfee860698584c030f4c9db1", topics: [
"0x0000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b", null, [ "0x0000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b",
"0x0000000000000000000000000aff3454fce5edbc8cca8697c15331677e6ebccc", ], ], }, ]
```

Returns `QUANTITY` - A filter id.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newFilter","params":[{"topics":["0x12341234"]}],"id":73}' // Result { "id":1, "jsonrpc": "2.0", "result": "0x1" //
1 }
```

eth_newBlockFilter {#eth_newblockfilter}

Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call eth_getFilterChanges.

Parameters None

Returns `QUANTITY` - A filter id.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newBlockFilter","params":[],"id":73}' // Result { "id":1, "jsonrpc": "2.0", "result": "0x1" // 1 }
```

eth_newPendingTransactionFilter {#eth_newpendingtransactionfilter}

Creates a filter in the node, to notify when new pending transactions arrive. To check if the state has changed, call eth_getFilterChanges.

Parameters None

Returns `QUANTITY` - A filter id.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newPendingTransactionFilter","params":[],"id":73}' // Result { "id":1, "jsonrpc": "2.0", "result": "0x1" // 1 }
```

eth_uninstallFilter {#eth_uninstallfilter}

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additionally Filters timeout when they aren't requested with eth_getFilterChanges for a period of time.

Parameters

1. `QUANTITY` - The filter id.

```
js params: [ "0xb", // 11 ]
```

Returns `Boolean` - `true` if the filter was successfully uninstalled, otherwise `false`.

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_uninstallFilter","params":["0xb"],"id":73}' // Result { "id":1, "jsonrpc": "2.0", "result": true }
```

eth_getFilterChanges {#eth_getfilterchanges}

Polling method for a filter, which returns an array of logs which occurred since last poll.

Parameters

1. `QUANTITY` - the filter id.

```
js params: [ "0x16", // 22 ]
```

Returns `Array` - Array of log objects, or an empty array if nothing has changed since last poll.

- For filters created with `eth_newBlockFilter` the return are block hashes (`DATA`, 32 Bytes), e.g. `["0x3454645634534..."]`.
- For filters created with `eth_newPendingTransactionFilter` the return are transaction hashes (`DATA`, 32 Bytes), e.g. `["0x6345343454645..."]`.
- For filters created with `eth_newFilter` logs are objects with following params:
- `removed`: `TAG` - `true` when the log was removed, due to a chain reorganization. `false` if its a valid log.
- `logIndex`: `QUANTITY` - integer of the log index position in the block. `null` when its pending log.
- `transactionIndex`: `QUANTITY` - integer of the transactions index position log was created from. `null` when its pending log.
- `transactionHash`: `DATA`, 32 Bytes - hash of the transactions this log was created from. `null` when its pending log.
- `blockHash`: `DATA`, 32 Bytes - hash of the block where this log was in. `null` when its pending. `null` when its pending log.
- `blockNumber`: `QUANTITY` - the block number where this log was in. `null` when its pending. `null` when its pending log.
- `address`: `DATA`, 20 Bytes - address from which this log originated.
- `data`: `DATA` - contains zero or more 32 Bytes non-indexed arguments of the log.
- `topics`: `Array of DATA` - Array of 0 to 4 32 Bytes `DATA` of indexed log arguments. (In *solidity*: The first topic is the *hash* of the signature of the event (e.g. `Deposit(address,bytes32,uint256)`), except you declared the event with the `anonymous` specifier.)
- **Example**

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getFilterChanges","params":["0x16"],"id":73}' // Result { "id":1, "jsonrpc":"2.0", "result": [{ "logIndex": "0x1",
// 1 "blockNumber":"0x1b4", // 436 "blockHash": "0x8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcfdf829c5a142f1fccd7d", "transactionHash":
"0xdf829c5a142f1fccd7d8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcf", "transactionIndex": "0x0", // 0 "address": "0x16c5785ac562ff41e2dcfdf829c5a142f1fccd7d",
"data":"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000", "topics": ["0x59ebeb90bc63057b6515673c3ecf9438e5058bca0f92585014eced636878c9a5"] },{ ... }] }
```

eth_getFilterLogs {#eth_getfilterlogs}

Returns an array of all logs matching filter with given id.

Parameters

1. `QUANTITY` - The filter id.

```
js params: [ "0x16", // 22 ]
```

Returns See eth_getFilterChanges

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getFilterLogs","params":["0x16"],"id":74}'
```

Result see eth_getFilterChanges

eth_getLogs {#eth_getlogs}

Returns an array of all logs matching a given filter object.

Parameters

1. `Object` - The filter options:
2. `fromBlock`: `QUANTITY|TAG` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
3. `toBlock`: `QUANTITY|TAG` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
4. `address`: `DATA|Array`, 20 Bytes - (optional) Contract address or a list of addresses from which logs should originate.
5. `topics`: `Array of DATA`, - (optional) Array of 32 Bytes `DATA` topics. Topics are order-dependent. Each topic can also be an array of DATA with "or" options.
6. `blockhash`: `DATA`, 32 Bytes - (optional, **future**) With the addition of EIP-234, `blockHash` will be a new filter option which restricts the logs returned to the single block with the 32-byte hash `blockHash`. Using `blockHash` is equivalent to `fromBlock` = `toBlock` = the block number with hash `blockHash`. If `blockHash` is present in the filter criteria, then neither `fromBlock` nor `toBlock` are allowed.

```
js params: [ { topics: [ "0x000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b", ], }, ]
```

Returns See eth_getFilterChanges

Example

```
js // Request curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getLogs","params":[{"topics":["0x000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b"]}],"id":74}'
```

Result see eth_getFilterChanges

Usage Example {#usage-example}

Deploying a contract using JSON_RPC {#deploying-contract}

This section includes a demonstration of how to deploy a contract using only the RPC interface. There are alternative routes to deploying contracts where this complexity is abstracted away—for example, using libraries built on top of the RPC interface such as web3.js and web3.py. These abstractions are generally easier to understand and less error-prone, but it is still helpful to understand what is happening under the hood.

The following is a straightforward smart contract called `Multiply7` that will be deployed using the JSON-RPC interface to an Ethereum node. This tutorial assumes the reader is already running a Geth node. More information on nodes and clients is available here. Please refer to individual client documentation to see how to start the HTTP JSON-RPC for non-Geth clients. Most clients default to serving on `localhost:8545`.

```
javascript contract Multiply7 { event Print(uint); function multiply(uint input) returns (uint) { Print(input * 7); return input * 7; } }
```

The first thing to do is make sure the HTTP RPC interface is enabled. This means we supply Geth with the `-http` flag on startup. In this example we use the Geth node on a private development chain. Using this approach we don't need ether on the real network.

```
bash geth --http --dev console 2>>geth.log
```

This will start the HTTP RPC interface on `http://localhost:8545`.

We can verify that the interface is running by retrieving the Coinbase address and balance using curl. Please note that data in these examples will differ on your local node. If you want to try these commands, replace the request params in the second curl request with the result returned from the first.

```bash curl --data '{"jsonrpc":"2.0","method":"eth_coinbase", "id":1}' -H "Content-Type: application/json" localhost:8545 {"id":1,"jsonrpc":"2.0","result":
["0x9b1d35635cc34752ca54713bb99d38614f63c955"]}

curl --data '{"jsonrpc":"2.0","method":"eth_getBalance", "params": ["0x9b1d35635cc34752ca54713bb99d38614f63c955", "latest"], "id":2}' -H "Content-Type: application/json" localhost:8545
{"id":2,"jsonrpc":"2.0","result":"0x1639e49bba16280000"} ```
```

Because numbers are hex encoded, the balance is returned in wei as a hex string. If we want to have the balance in ether as a number we can use web3 from the Geth console.

```
javascript web3.fromWei("0x1639e49bba16280000", "ether") // "410"
```

Now that there is some ether on our private development chain, we can deploy the contract. The first step is to compile the Multiply7 contract to byte code that can be sent to the EVM. To install solc, the Solidity compiler, follow the [Solidity documentation](). (You might want to use an older `solc` release to match [the version of compiler used for our example]())

The next step is to compile the Multiply7 contract to byte code that can be send to the EVM.

```bash echo 'pragma solidity ^0.4.16; contract Multiply7 { event Print(uint); function multiply(uint input) public returns (uint) { Print(input * 7); return input * 7; } }' | solc --bin

======= :Multiply7 ======= Binary:
6060604052341561000f57600080fd5b60eb8061001d6000396000f300606060405260043610603f576000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff16806
```

Now that we have the compiled code we need to determine how much gas it costs to deploy it. The RPC interface has an `eth_estimateGas` method that will give us an estimate.

```
bash curl --data '{"jsonrpc":"2.0","method": "eth_estimateGas", "params": [{"from": "0x9b1d35635cc34752ca54713bb99d38614f63c955", "data":
"0x6060604052341561000f57600080fd5b60eb8061001d6000396000f300606060405260043610603f576000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff168063c6888fa11460
"id": 5}' -H "Content-Type: application/json" localhost:8545 {"jsonrpc":"2.0","id":5,"result":"0x1c31e"}
```

And finally deploy the contract.

```
bash curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction", "params": [{"from": "0x9b1d35635cc34752ca54713bb99d38614f63c955", "gas": "0x1c31e", "data":
"0x6060604052341561000f57600080fd5b60eb8061001d6000396000f300606060405260043610603f576000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff168063c6888fa11460
"id": 6}' -H "Content-Type: application/json" localhost:8545 {"id":6,"jsonrpc":"2.0","result":"0xe1f3095770633ab2b18081658bad475439f6a08c902d0915903bafff06e6febf"}
```

The transaction is accepted by the node and a transaction hash is returned. This hash can be used to track the transaction. The next step is to determine the address where our contract is deployed. Each executed transaction will create a receipt. This receipt contains various information about the transaction such as in which block the transaction was included and how much gas was used by the EVM. If a transaction creates a contract it will also contain the contract address. We can retrieve the receipt with the `eth_getTransactionReceipt` RPC method.

```
bash curl --data '{"jsonrpc":"2.0","method": "eth_getTransactionReceipt", "params": ["0xe1f3095770633ab2b18081658bad475439f6a08c902d0915903bafff06e6febf"], "id": 7}' -H "Content-
Type: application/json" localhost:8545 {"jsonrpc":"2.0","id":7,"result":
{"blockHash":"0x77b1a4f6872b9066312de3744f60020cbd8102af68b1f6512a05b7619d527a4f","blockNumber":"0x1","contractAddress":"0x4d03d617d700cf81935d7f797f4e2ae719648262","cumulativeGasUsed
[],"logsBloom":"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Our contract was created on `0x4d03d617d700cf81935d7f797f4e2ae719648262`. A null result instead of a receipt means the transaction has not been included in a block yet. Wait for a moment and check if your miner is running and retry it.

Interacting with smart contracts {#interacting-with-smart-contract}

In this example we will be sending a transaction using `eth_sendTransaction` to the `multiply` method of the contract.

`eth_sendTransaction` requires several arguments, specifically `from`, `to` and `data`. `From` is the public address of our account, and `to` is the contract address. The `data` argument contains a payload that defines which method must be called and with which arguments. This is where the [ABI (application binary interface)]() comes into play. The ABI is a JSON file that defines how to define and encode data for the EVM.

The bytes of the payload defines which method in the contract is called. This is the first 4 bytes from the Keccak hash over the function name and its argument types, hex encoded. The multiply function accepts an uint which is an alias for uint256. This leaves us with:

```
javascript web3.sha3("multiply(uint256)").substring(0, 10) // "0xc6888fa1"
```

The next step is to encode the arguments. There is only one uint256, say, the value 6. The ABI has a section which specifies how to encode uint256 types.

`int<M>: enc(X)` is the big-endian two's complement encoding of X, padded on the higher-order (left) side with 0xff for negative X and with zero > bytes for positive X such that the length is a multiple of 32 bytes.

This encodes to `0000000000000000000000000000000000000000000000000000000000000006`.

Combining the function selector and the encoded argument our data will be `0xc6888fa10000000000000000000000000000000000000000000000000000000000000006`.

This can now be sent to the node:

```
bash curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction", "params": [{"from": "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", "to":
"0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d", "data": "0xc6888fa10000000000000000000000000000000000000000000000000000000000000006"}], "id": 8}' -H "Content-Type: application/json"
localhost:8545 {"id":8,"jsonrpc":"2.0","result":"0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74"}
```

Since a transaction was sent, a transaction hash was returned. Retrieving the receipt gives:

```
javascript { blockHash: "0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55", blockNumber: 268, contractAddress: null, cumulativeGasUsed: 22631, gasUsed: 22631,
logs: [{ address: "0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d", blockHash: "0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55", blockNumber: 268, data:
"0x000000000000000000000000000000000000000000000000000000000000002a", logIndex: 0, topics: ["0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"], transactionHash:
"0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74", transactionIndex: 0 }], transactionHash: "0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
transactionIndex: 0 }
```

The receipt contains a log. This log was generated by the EVM on transaction execution and included in the receipt. The `multiply` function shows that the `Print` event was raised with the input times 7. Since the argument for the `Print` event was a uint256 we can decode it according to the ABI rules which will leave us with the expected decimal 42. Apart from the data it is worth noting that topics can be used to determine which event created the log:

```
javascript web3.sha3("Print(uint256)") // "24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
```

This was just a brief introduction into some of the most common tasks, demonstrating direct usage of the JSON-RPC.

Related topics {#related-topics}

- [JSON-RPC specification]()
- [Nodes and clients]()
- [JavaScript APIs]()
- [Backend APIs]()
- [Execution clients]()