Ethereum Metamask Browser Methods

Basic Considerations

#

Web3 Browser Detection

To verify if the browser is running MetaMask, copy and paste the code snippet below in the developer console of your web browser:

```
if (typeof window.ethereum !== 'undefined') {
  console.log('MetaMask is installed!');
}
```

You can review the full API for the `window.ethereum` object here.

#

Running a Test Network

In the top right menu of MetaMask, select the network that you are currently connected to. Among several popular defaults, you'll find `Custom RPC` and `Localhost 8545`. These are both useful for connecting to a test blockchain, like ganache

(opens new window)

. You can quickly install and start Ganache if you have `npm` installed with `npm i -g ganache-cli && ganache-cli`.

Ganache has some great features for starting your application with different states. If your application starts with the `-m` flag, you can feed it the same seed phrase you have in your MetaMask, and the test network will give each of your first 10 accounts 100 test ether, which makes it easier to start work.

Since your seed phrase has the power to control all your accounts, it is probably worth keeping at least one seed phrase for development, separate from any that you use for storing real value. One easy way to manage multiple seed phrases with MetaMask is with multiple browser profiles, each of which can have its own clean extension installations.

Resetting Your Local Nonce Calculation

If you're running a test blockchain and restart it, you can accidentally confuse MetaMask because it calculates the next nonce based on both the network state *and* the known sent transactions.

To clear MetaMask's transaction queue, and effectively reset its nonce calculation, you can use the `Reset Account` button in `Settings` (available in the top-right identicon menu).

Detecting MetaMask

If you want to differentiate MetaMask from other Ethereum-compatible browsers, you can detect MetaMask using `ethereum.isMetaMask`.

User State

Currently there are a few stateful things to consider when interacting with this API:

- What is the current network?

- What is the current account?

Both of these are available synchronously as `ethereum.networkVersion` and `ethereum.selectedAddress`. You can listen for changes using events too, see (the API reference).

#

Connecting to MetaMask

"Connecting" or "logging in" to MetaMask effectively means "to access the user's Ethereum account(s)".

You should only initiate a connection request in response to direct user action, such as clicking a button. You should always disable the "connect" button while the connection request is pending. You should never initiate a connection request on page load.

We recommend that you provide a button to allow the user to connect MetaMask to your dapp. Clicking this button should call the following method:

```
ethereum.request({ method: 'eth_requestAccounts' });
```

Example:

```
const ethereumButton = document.querySelector('.enableEthereumButton');


ethereumButton.addEventListener('click', () => {
  //Will Start the metamask extension
  ethereum.request({ method: 'eth_requestAccounts' });
```

});

This promise-returning function resolves with an array of hex-prefixed Ethereum addresses, which can be used as general account references when sending transactions.

Over time, this method is intended to grow to include various additional parameters to help your site request everything it needs from the user during setup.

Since it returns a promise, if you're in an `async` function, you may log in like this:

```
const accounts = await ethereum.request({ method: 'eth_requestAccounts' });

const account = accounts[0];

// We currently only ever provide a single account,

// but the array gives us some room to grow.
```

Example:

```
const ethereumButton = document.querySelector('.enableEthereumButton');

const showAccount = document.querySelector('.showAccount');


ethereumButton.addEventListener('click', () => {

  getAccount();

});
```

```
async function getAccount() {
  const accounts = await ethereum.request({ method: 'eth_requestAccounts' });
  const account = accounts[0];
  showAccount.innerHTML = account;
}
```

Choosing a Convenience Library

Convenience libraries exist for a variety of reasons.

Some of them simplify the creation of specific user interface elements, some entirely manage the user account onboarding, and others give you a variety of methods of interacting with smart contracts, for a variety of API preferences, from promises, to callbacks, to strong types, and on.

The provider API itself is very simple, and wraps Ethereum JSON-RPC

(opens new window)

formatted messages, which is why developers usually use a convenience library for interacting with the provider, like ethers

(opens new window)

, web3.js

(opens new window)

, truffle

(opens new window)

, Embark

(opens new window)

, or others. From those tools, you can generally find sufficient documentation to interact with the provider, without reading this lower-level API.

MetaMask injects a global API into websites visited by its users at `window.ethereum`. This API allows websites to request users' Ethereum accounts, read data from blockchains the user is connected to, and suggest that the user sign messages and transactions. The presence of the provider object indicates an Ethereum user. We recommend using

`@metamask/detect-provider`

(opens new window)

to detect our provider, on any platform or browser.

The Ethereum JavaScript provider API is specified by EIP-1193

```
// This function detects most providers injected at window.ethereum
import detectEthereumProvider from '@metamask/detect-provider';
```

```
const provider = await detectEthereumProvider();

if (provider) {
  // From now on, this should always be true:
  // provider === window.ethereum
  startApp(provider); // initialize your app
} else {
  console.log('Please install MetaMask!');
}
```

Table of Contents

#

Basic Usage

For any non-trivial Ethereum web application — a.k.a. dapp, web3 site etc. — to work, you will have to:

- Detect the Ethereum provider (`window.ethereum`)
- Detect which Ethereum network the user is connected to
- Get the user's Ethereum account(s)

The snippet at the top of this page is sufficient for detecting the provider. You can learn how to accomplish the other two by reviewing the snippet in the Using the Provider section.

The provider API is all you need to create a full-featured web3 application.

That said, many developers use a convenience library, such as ethers

(opens new window)

, instead of using the provider directly. If you are in need of higher-level abstractions than those provided by this API, we recommend that you use a convenience library.

#

Chain IDs

These are the IDs of the Ethereum chains that MetaMask supports by default. Consult chainid.network

(opens new window)

for more.

Hex	Decimal	Network
0x1	1	Ethereum Main Network (Mainnet)
0x3	3	Ropsten Test Network
0x4	4	Rinkeby Test Network
0x5	5	Goerli Test Network
0x2a	42	Kovan Test Network

#

Properties

#

ethereum.isMetaMask

Note

This property is non-standard. Non-MetaMask providers may also set this property to `true`.

`true` if the user has MetaMask installed.

#

Methods

#

ethereum.isConnected()

Tip

Note that this method has nothing to do with the user's accounts.

You may often encounter the word "connected" in reference to whether a web3 site can access the user's accounts. In the provider interface, however, "connected" and "disconnected" refer to whether the provider can make RPC requests to the current chain.

```
const isConnected = ethereum.isConnected();
```

Returns `true` if the provider is connected to the current chain, and `false` otherwise.

If the provider is not connected, the page will have to be reloaded in order for connection to be re-established. Please see the `connect` and `disconnect` events for more information.

#

ethereum.request(args)

```
const requestArguments = {
  method: "exampleMethod",
  params: [param1, param2, param3]
};

ethereum.request(requestArguments)
  .then((result) => {
    // Handle the result here
    console.log(result);
  })
  .catch((error) => {
    // Handle the error here
    console.error(error);
  });
```

Use `request` to submit RPC requests to Ethereum via MetaMask. It returns a `Promise` that resolves to the result of the RPC method call.

The `params` and return value will vary by RPC method. In practice, if a method has any `params`, they are almost always of type `Array<any>`.

If the request fails for any reason, the Promise will reject with an Ethereum RPC Error.

MetaMask supports most standardized Ethereum RPC methods, in addition to a number of methods that may not be supported by other wallets. See the MetaMask RPC API documentation for details.

#

Example

```
const params = [
  {
    from: '0xb60e8dd61c5d32be8058bb8eb970870f07233155',
    to: '0xd46e8dd67c5d32be8058bb8eb970870f07244567',
    gas: '0x76c0', // 30400
    gasPrice: '0x9184e72a000', // 10000000000000
    value: '0x9184e72a', // 2441406250
    data:

'0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f07
2445675',
  },
];

ethereum
  .request({
    method: 'eth_sendTransaction',
    params: params,
  })
  .then((result) => {
    // The result varies by RPC method.
    // For example, this method will return a transaction hash hexadecimal string on success.
    console.log(result);
  })
  .catch((error) => {
    // If the request fails, the Promise will reject with an error.
    console.error(error);
  });
```

Events

The MetaMask provider implements the Node.js `EventEmitter`

(opens new window)

API. This sections details the events emitted via that API. There are innumerable

`EventEmitter` guides elsewhere, but you can listen for events like this:

```
ethereum.on('accountsChanged', (accounts) => {
  // Handle the new accounts, or lack thereof.
  // "accounts" will always be an array, but it can be empty.
  console.log(accounts);
});

ethereum.on('chainChanged', (chainId) => {
  // Handle the new chain.
  // Correctly handling chain changes can be complicated.
  // We recommend reloading the page unless you have good reason not to.
  window.location.reload();
});
```

Also, don't forget to remove listeners once you are done listening to them (for example on component unmount in React):

```
function handleAccountsChanged(accounts) {
  // ...
}

ethereum.on('accountsChanged', handleAccountsChanged);

// Later

ethereum.removeListener('accountsChanged', handleAccountsChanged);
```

The first argument of the `ethereum.removeListener` is the event name and the second argument is the reference to the same function which has passed to `ethereum.on` for the event name mentioned in the first argument.

#

connect

```
ethereum.on('connect', (connectInfo) => {
  // Handle the connect event
  console.log(connectInfo.chainId);
});
```

The MetaMask provider emits this event when it first becomes able to submit RPC requests to a chain. We recommend using a `connect` event handler and the `ethereum.isConnected()` method in order to determine when/if the provider is connected.

#

disconnect

```
ethereum.on('disconnect', (error) => {
  // Handle the disconnect event
  console.log(error);
});
```

The MetaMask provider emits this event if it becomes unable to submit RPC requests to any chain. In general, this will only happen due to network connectivity issues or some unforeseen error.

Once `disconnect` has been emitted, the provider will not accept any new requests until the connection to the chain has been re-established, which requires reloading the page. You can also use the `ethereum.isConnected()` method to determine if the provider is disconnected.

#

accountsChanged

```
ethereum.on('accountsChanged', (accounts) => {
  // Handle the accountsChanged event
  console.log(accounts);
});
```

The MetaMask provider emits this event whenever the return value of the `eth_accounts` RPC method changes. `eth_accounts` returns an array that is either empty or contains a single account address. The returned address, if any, is the address of the most recently used account that the caller is permitted to access. Callers are identified by their URL *origin*, which means that all sites with the same origin share the same permissions.

This means that `accountsChanged` will be emitted whenever the user's exposed account address changes.

> Tip
>
> We plan to allow the `eth_accounts` array to be able to contain multiple addresses in the near future.

#

chainChanged

> Tip
>
> See the Chain IDs section for MetaMask's default chains and their chain IDs.

```
ethereum.on('chainChanged', (chainId) => {
  // Handle the chainChanged event
  console.log(chainId);
});
```

The MetaMask provider emits this event when the currently connected chain changes.

All RPC requests are submitted to the currently connected chain. Therefore, it's critical to keep track of the current chain ID by listening for this event.

We *strongly* recommend reloading the page on chain changes, unless you have good reason not to.

```
ethereum.on('chainChanged', (_chainId) => {
  // Handle the chainChanged event
```

message

```
ethereum.on('message', (message) => {
  // Handle the message event
  console.log(message);
});
```

The MetaMask provider emits this event when it receives some message that the consumer should be notified of. The kind of message is identified by the `type` string.

RPC subscription updates are a common use case for the `message` event. For example, if you create a subscription using `eth_subscribe`, each subscription update will be emitted as a `message` event with a `type` of `eth_subscription`.

#

Errors

All errors thrown or returned by the MetaMask provider follow this interface:

```
class ProviderRpcError extends Error {
  constructor(message, code, data) {
    super(message);
    this.code = code;
    this.data = data;
    this.name = 'ProviderRpcError';
  }
}
```

The `ethereum.request(args)` method throws errors eagerly. You can often use the error `code` property to determine why the request failed. Common codes and their meaning include:

- `4001`
  - The request was rejected by the user
- `-32602`
  - The parameters were invalid
- `-32603`
  - Internal error

For the complete list of errors, please see EIP-1193

(opens new window)

and EIP-1474

(opens new window)

.

Tip

The `eth-rpc-errors`

(opens new window)

package implements all RPC errors thrown by the MetaMask provider, and can help you identify their meaning.

#

Using the Provider

This snippet explains how to accomplish the three most common requirements for web3 sites:

- Detect the Ethereum provider (`window.ethereum`)
- Detect which Ethereum network the user is connected to
- Get the user's Ethereum account(s)

```
import detectEthereumProvider from '@metamask/detect-provider';

async function startApp(provider) {
  if (provider !== window.ethereum) {
    console.error('Do you have multiple wallets installed?');
  }
  // Access the decentralized web!
}

async function handleChainChanged(_chainId) {
  // We recommend reloading the page, unless you must do otherwise
  window.location.reload();
}

let currentAccount = null;

async function handleAccountsChanged(accounts) {
  if (accounts.length === 0) {
    // MetaMask is locked or the user has not connected any accounts
    console.log('Please connect to MetaMask.');
  } else if (accounts[0] !== currentAccount) {
    currentAccount = accounts[0];
    // Do any other work!
  }
}

async function connect() {
  try {
    const accounts = await ethereum.request({ method: 'eth_requestAccounts' });
    handleAccountsChanged(accounts);
  } catch (err) {
    if (err.code === 4001) {
      // EIP-1193 userRejectedRequest error
      // If this happens, the user rejected the connection request.
```

```
      console.log('Please connect to MetaMask.');
    } else {
      console.error(err);
    }
  }
}

(async () => {
  const provider = await detectEthereumProvider();

  if (provider) {
    startApp(provider); // Initialize your app
  } else {
    console.log('Please install MetaMask!');
  }

  const chainId = await ethereum.request({ method: 'eth_chainId' });
  handleChainChanged(chainId);

  ethereum.on('chainChanged', handleChainChanged);

  try {
    const accounts = await ethereum.request({ method: 'eth_accounts' });
    handleAccountsChanged(accounts);
  } catch (err) {
    console.error(err);
  }

  ethereum.on('accountsChanged', handleAccountsChanged);

  document.getElementById('connectButton').addEventListener('click', connect);
})();
```