Make a Transfer

A transfer involves one wallet sending tokens to another wallet on the OMG Network.

Implementation

1. Install `omg-js`, `bn.js`

To access network features from your application, use our official libraries:

Node Browser React Native

Requires Node >= 8.11.3 < 13.0.0

```
npm install @omisego/omg-js bn.js
```

JavaScript (ESNext)

2. Import dependencies, define constants

Transferring funds to the OMG Network involves using ChildChain and OmgUtil `omg-js` objects. Here's an example of how to instantiate them:

```
import BigNumber from "bn.js";
import { ChildChain, OmgUtil } from "@omisego/omg-js";

const plasmaContractAddress = plasmaContractAddress;

// instantiate omg-js objects
const childChain = new ChildChain({
  watcherUrl: watcherUrl,
  watcherProxyUrl: '',
  plasmaContractAddress: plasmaContractAddress
});

// define constants
// amount => 0.012 with 18 decimals (ETH) = 12000000000000000
const ethTransfer = {
  sender: "0x8CB0DE6206f459812525F2BA043b14155C2230C0",
  senderPrivateKey: "CD55F2A7C476306B27315C7986BC50BD81DB4130D4B5CFD49E3EAF9ED1EDE4F7",
  receiver: "0xA9cc140410c2bfEB60A7260B3692dcF29665c254",
  currency: OmgUtil.transaction.ETH_CURRENCY,
  feeCurrency: OmgUtil.transaction.ETH_CURRENCY,
  amount: new BigNumber("12000000000000000"),
  metadata: "eth transfer"
}

// amount => 3 with 6 decimals (TUSDT) = 3000000
const erc20Transfer = {
  sender: "0x8CB0DE6206f459812525F2BA043b14155C2230C0",
  senderPrivateKey: "CD55F2A7C476306B27315C7986BC50BD81DB4130D4B5CFD49E3EAF9ED1EDE4F7",
  receiver: "0xA9cc140410c2bfEB60A7260B3692dcF29665c254",
  currency: OmgUtil.hexPrefix("0xd92e713d051c37ebb2561803a3b5fbabc4962431"),
  feeCurrency: OmgUtil.transaction.ETH_CURRENCY,
  amount: new BigNumber("3000000"),
  metadata: "TUSDT transfer"
}
```

- `watcherUrl` - the Watcher Info URL for defined environment (personal or from OMG Network).
- `plasmaContractAddress` - `CONTRACT_ADDRESS_PLASMA_FRAMEWORK` for defined environment.

There are several ways to send a transaction on the OMG Network. It's recommended to use the first method but you may want to choose another approach for your specific use case.

3. Send a payment transaction

Transactions are composed of inputs and outputs. An input is simply a pointer to the output of another transaction. An output is a transaction that hasn't been spent yet (also known as UTXO). Each transaction should be signed by the owner of funds (UTXOs), have a specific format, and encoded with RLP encoding according to the following rules:

Copy

```
[txType, inputs, outputs, txData, metaData]

txType ::= uint256
inputs ::= [input]
input ::= bytes32
outputs ::= [output]
output ::= [outputType, outputData]
outputType ::= uint256
outputData ::= [outputGuard, token, amount]
outputGuard ::= bytes20
token ::= bytes20
amount ::= uint256
txData ::= uint256 (must be 0)
metadata ::= bytes32
```

Transactions are signed using the EIP-712 method. The EIP-712 typed data structure is defined as follows:

Copy

```
{
  types: {
    EIP712Domain: [
        { name: 'name', type: 'string' },
        { name: 'version', type: 'string' },
        { name: 'verifyingContract', type: 'address' },
        { name: 'salt', type: 'bytes32' }
    ],
    Transaction: [
        { name: 'txType', type: 'uint256' },
        { name: 'input0', type: 'Input' },
        { name: 'input1', type: 'Input' },
        { name: 'input2', type: 'Input' },
        { name: 'input3', type: 'Input' },
        { name: 'output0', type: 'Output' },
        { name: 'output1', type: 'Output' },
        { name: 'output2', type: 'Output' },
        { name: 'output3', type: 'Output' },
        { name: 'txData', type: 'uint256' },
        { name: 'metadata', type: 'bytes32' }
    ],
    Input: [
        { name: 'blknum', type: 'uint256' },
        { name: 'txindex', type: 'uint256' },
        { name: 'oindex', type: 'uint256' }
    ],
    Output: [
        { name: 'outputType', type: 'uint256' },
        { name: 'outputGuard', type: 'bytes20' },
        { name: 'currency', type: 'address' },
        { name: 'amount', type: 'uint256' }
    ]
  },
  domain: {
        name: 'OMG Network',
        version: '1',
        verifyingContract: '',
        salt: '0xfad5c7f626d80f9256ef01929f3beb96e058b8b4b0e3fe52d84f054c0e2a7a83'
    },
  primaryType: 'Transaction'
}
```

Note, the child chain server collects fees for sending a transaction. The fee can be paid in a variety of supported tokens by the network. To get more details on how the fees are defined, please refer to Fees.

3.1 Method A

The most "granular" implementation of transfer includes creating, typing, signing, and submitting the transaction. Such an approach will have the following structure of the code:

This method demonstrates a transfer made in ETH. If you want to make an ERC20 transfer, replace `erc20Transfer` with the corresponding `ethTransfer` values.

```
async function transfer() {
  // construct a transaction body
  const transactionBody = await childChain.createTransaction({
    owner: erc20Transfer.sender,
    payments: [
      {
        owner: erc20Transfer.receiver,
        currency: erc20Transfer.currency,
        amount: erc20Transfer.amount,
      },
    ],
    fee: {
      currency: erc20Transfer.feeCurrency,
    },
    metadata: erc20Transfer.metadata,
  });

  // sanitize transaction into the correct typedData format
  // the second parameter is the address of the Plasma RootChain contract
  const typedData = OmgUtil.transaction.getTypedData(
    transactionBody.transactions[0], plasmaContractAddress);

  // define private keys to use for transaction signing
  const privateKeys = new Array(
    transactionBody.transactions[0].inputs.length
  ).fill(ethTransfer.senderPrivateKey);

  // locally sign typedData with passed private keys, useful for multiple different signatures
  const signatures = childChain.signTransaction(typedData, privateKeys);

  // return encoded and signed transaction ready to be submitted
  const signedTypedData = childChain.buildSignedTransaction(typedData, signatures);

  // submit to the child chain
  const receipt = await childChain.submitTransaction(signedTypedData);
  return receipt;
}
```

Additional Notes

The current technical implementation of the `createTransaction` function can return two possible responses:

1. If your transfer can be covered with four inputs or less, you'll create a payment transaction with `"result": "complete"` response.
2. If your transfer can be covered but with more than four inputs, you'll create a merge transaction with `"result": "intermediate"` response.

This is a temporary issue and will be fixed over time, however, you still need to be aware of this behavior and create an additional check for `result` status after calling the `createTransaction` function. Note, that `"result": "intermediate"` response doesn't re-create a payment transaction in this scenario, thus you'll need to initiate your original transaction again after the merge transaction is confirmed.

For more advanced types of transactions, please refer to Make a Fee Relay Transfer or Make an Atomic Swap.

Lifecycle

1. A user calls the `createTransaction` function to create a transaction.
2. A user signs, encodes, and submits the transaction's data to the child chain and the Watcher for validation.
3. If the transaction is valid, the child chain server creates a transaction hash and adds the transaction to a pending block.
4. The child chain bundles the transactions in the block into a Merkle tree and submits its root hash to the `Plasma Framework` contract.
5. The Watcher receives a list of transactions from the child chain and recomputes the Merkle root to check for any inconsistency.

The following conditions would cause the Watcher or the child chain to reject the transaction as invalid:

- The transaction is using inputs used for another transaction in the block.
- The transaction is using inputs spent in any prior block.
- The transaction is using inputs that were exited.
- The transaction is using inputs from a non-validated deposit.
- The transaction is signed with an invalid signature.

Demo Project

This section provides a demo project that contains a detailed implementation of the tutorial. If you consider integrating with the OMG Network, you can use this sample to significantly reduce the time of development. It also provides step-by-step instructions and sufficient code guidance that is not covered on this page.

JavaScript

For running a full `omg-js` code sample for the tutorial, please use the following steps:

1. Clone omg-js-samples repository:

```
git clone https://github.com/omgnetwork/omg-js-samples.git
```

2. Create `.env` file and provide the required configuration values.

3. Run these commands:

```
npm install
npm run start
```

4. Open your browser at http://localhost:3000.

5. Select `Make an ETH Transaction` or `Make an ERC20 Transaction` on the left side, observe the logs on the right.

Code samples for all tutorials use the same repository — `omg-js-samples`, thus you have to set up the project and install dependencies only one time.