Ethers.js Quick Guide

Contents

What is Ethers.js

Ethers.js is a library for interacting with the Ethereum Blockchain

Installation and usage:

```
npm i ethers --save
```

```
const { ethers } = require("ethers"); //node.js
import { ethers } from "ethers"; //ES6
```

Important terms:

Provider: A class that provides a connection to the Ethereum Network - enables read-only access to the Blockchain.

Signer: A class that has access to a private key (in order to authorize the network to charge your account ether to execute transactions) and can sign messages and transactions.

Contract: A class that provides a connection to a specific contract on the Ethereum Network.

Complete documentation: https://docs.ethers.io/v5/

Quick overview

Connecting to Ethereum via Metamask:

The quickest and easiest way to begin developing on Ethereum is to use MetaMask, which is a browser extension that provides a connection to the Ethereum network (a Provider) and holds your private key in order to sign transactions (a Signer).

```
// A Web3Provider wraps a standard Web3 provider, which is
// what MetaMask injects as window.ethereum into each page
const provider = new ethers.providers.Web3Provider(window.ethereum)

// Metamask Json-RPC method: Requests (popup) that the user authorizes
// an address for the application
await provider.send("eth_requestAccounts", []);

const signer = provider.getSigner()
```

Connecting to Ethereum via RPC:

The JSON-RPC API is another popular method for interacting with Ethereum. This is available for various third-party services like Infura, Alchemy...

```
// If you don't have a url, Ethers connects to the default - i.e. http://localhost:8545
const provider = new ethers.providers.JsonRpcProvider();
```

Querying the blockchain:

```
balance = await provider.getBalance("0x123...")
ethers.utils.formatEther(balance)
ethers.utils.parseEther("1.0")
```

Writing to the Blockchain:

```
// Send 1 ether to an ens name.
const tx = signer.sendTransaction({
   to: "0x123...",
   value: ethers.utils.parseEther("1.0")
});
```

Contracts

In order to communicate with the Contract on-chain, this class needs to know what methods are available => it gets this information from the ABI.

```
const myContract = new ethers.Contract(contractAddress, ABI, provider);
```

Read-only methods:

```
const result = await myContract.someMethod()
```

State changing methods

```
// The contract is currently connected to the provider,
// which is read only => connect to a signer to perform transactions
const myContractWithSigner = myContract.connect(signer)
const txn = await myContractWithSigner.someWriteMethod(55)
```

Logs & filtering - listen to event:

Logs and filtering are used quite often in blockchain applications, since they allow for efficient queries of indexed data and provide lower-cost data storage when the data is not required to be accessed on-chain. These can be used in conjunction with the Provider Events API and with the Contract Events API.

When a Contract creates a log, it can include up to 4 pieces of data to be indexed by. The indexed data is hashed and included in a Bloom Filter, which is a data structure that allows for efficient filtering.

```
myContract.on("EventName", (from, to) => {
   console.log(`${ from } sent ${to}`);
});

// query historic events
myAddress = await signer.getAddress()

// Filter for all token transfers from me
filterFrom = myContract.filters.Transfer(myAddress, null);

// Filter for all token transfers to me
filterTo = myContract.filters.Transfer(null, myAddress);

// List all transfers sent from me in a specific block range
await myContract.queryFilter(filterFrom, 9843470, 9843480)

// List all transfers sent in the last 10,000 blocks
await myContract.queryFilter(filterFrom, -10000)

// List all transfers ever sent to me
await myContract.queryFilter(filterTo)
```

Signing a message

```
// To sign a simple string, which are used for
// logging into a service, pass the string in.
signature = await signer.signMessage("My Message");

// A common case is also signing a hash, which is 32
// bytes. To sign binary data it MUST be an Array
message = "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"
messageBytes = ethers.utils.arrayify(message);
signature = await signer.signMessage(messageBytes)
```

Handling a network change:

```
// Force page refreshes on network changes
{
   // The "any" network will allow spontaneous network changes
   const provider = new ethers.providers.Web3Provider(window.ethereum, "any");
   provider.on("network", (newNetwork, oldNetwork) => {
      // When a Provider makes its initial connection, it emits a "network"
```

```
        // event with a null oldNetwork along with the newNetwork. So, if the
        // oldNetwork exists, it represents a changing network
        if (oldNetwork) {
            window.location.reload();
        }
    });
}
```

Provider API keys

The ethers library offers default API keys for each service, so that each Provider works out-of-the-box. These API keys are provided for low-traffic projects and for early prototyping.

The default provider connects to multiple backends and verifies their results internally.
A second optional parameter allows API keys to be specified to each Provider created

```
const network = "homestead";

// Specify your own API keys - each is optional, and if you omit it the default
// API key for that service will be used.
const provider = ethers.getDefaultProvider(network, {
    etherscan: YOUR_ETHERSCAN_API_KEY,
    infura: YOUR_INFURA_PROJECT_ID,
    // Or if using a project secret:
    // infura: {
    //   projectId: YOUR_INFURA_PROJECT_ID,
    //   projectSecret: YOUR_INFURA_PROJECT_SECRET,
    // },
    alchemy: YOUR_ALCHEMY_API_KEY
});

// Usage without an API key
const provider = ethers.getDefaultProvider((network = "goerli"));
```

Providers

A Provider is an abstraction of a connection to the Ethereum network. The default provider is the safest, easiest way to begin developing on Ethereum. It creates a FallbackProvider connected to as many backend services as possible. When a request is made, it is sent to multiple backends simultaneously. A Provider in ethers is a read-only abstraction to access the blockchain data, a signer provides read/write access.

DefaultProvider

ethers.getDefaultProvider([network , [options]]) ⇒ Provider network: mainnet, goerli, http://localhost:8545...

The default API Keys used by ethers are shared across all users, so services may throttle all services

Provider methods:

In the ethers API, nearly anywhere that accepts an address, an ENS name may be used instead

```
await provider.getBalance("0x123...");
await provider.getTransactionCount("0x123..."); //nonce for the next txn
await provider.lookupAddress("0x123...") //get ENS name
await provider.resolveName("ricmoo.eth"); //get address

await provider.getLogs(filter) // Returns array of Log matching the filter
await provider.getNetwork() //returns chainId and name
await provider.getBlockNumber() //get most recently mined block

gasPrice = await provider.getGasPrice() //estimation of gas price in wei
utils.formatUnits(gasPrice, "gwei")
await provider.getFeeData() //estimation for maxFeePerGas, maxPriorityFeePerGas
```

Transaction methods:

```
//execute a txn using call (cannot change state) => call read-only functions
await provider.call({
  to: "0x4976fb03C32e5B8cfe2b6cCB31c09Ba78EBaBa41",
  // "function someFuntion(uint) view returns (address)`
  data: "0x3b3b57debf07..."
});

//execute a read/write txn - needs to be signed
const signedTx = "0xf86904018252...";
await provider.sendTransaction(signedTx);

//estimate gas for specific txn
await provider.estimateGas({
  to: "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2",
  // `function deposit() payable`
  data: "0xd0e30db0",
  value: parseEther("1.0")
});
```

Event methods:

```
provider.on( eventName , listener ) //Add a listener to be triggered for each event.
provider.off( eventName [ , listener ] ) //If no listener provided, remove all listeners
```

```
provider.removeAllListeners( [ eventName ] )

provider.on("block", (blockNumber) => {
   // Emitted on every block change
})


provider.once(txHash, (transaction) => {
   // Emitted when the transaction has been mined
})


// This filter could also be generated with the Contract or
// Interface API. If address is not specified, any address
// matches and if topics is not specified, any log matches
filter = {
   address: "dai.tokens.ethers.eth",
   topics: [
      utils.id("Transfer(address,address,uint256)")
   ]
}
provider.on(filter, (log, event) => {
   // Emitted whenever a DAI token transfer occurs
})


// Notice this is an array of topic-sets and is identical to
// using a filter with no address (i.e. match any address)
topicSets = [
   utils.id("Transfer(address,address,uint256)"),
   null,
   [
      hexZeroPad(myAddress, 32),
      hexZeroPad(myOtherAddress, 32)
   ]
]
provider.on(topicSets, (log, event) => {
   // Emitted any token is sent TO either address
})
```

JsonRpcProvider

The JSON-RPC API is a popular method for interacting with Ethereum and is available in all major nodes (Parity, Geth) and third-party services: Infura, Alchemy...

```
new ethers.providers.JsonRpcProvider( [ urlOrConnectionInfo [ , networkish ] ] )

//Returns JsonRpcSigner at addressOrIndex, otherwise, first account is used
jsonRpcProvider.getSigner( [ addressOrIndex ] )
```

```
jsonRpcProvider.listAccounts( ) => Promise< Array< string > >
```

```
signer.sendUncheckedTransaction( transaction )
```

API Providers

There are providers for Etherscan, Infura, Alchemy, Cloudflre, Pocket and Ankr. However, reliance on third-party services can reduce resilience, security and increase the amount of required trust. To mitigate these issues, it is recommended you use a Default Provider.

```
//Network can be string or chainId, if no apiKey is provided, a shared API key will be used
//networks: homestead, rinkeby, goerli, matic, maticmum, optimism, arbitrum...
new ethers.providers.AlchemyProvider( [ network = "homestead" , [ apiKey ] ] )
```

```
// Connect to mainnet (homestead)
provider = new AlchemyProvider();
```

FallbackProvider

The FallbackProvider is the most advanced Provider. It uses a quorum and connects to multiple Providers, each configured with a priority and a weight.

The WebSocketProvider connects to a JSON-RPC WebSocket-compatible backend which allows for a persistent connection, multiplexing requests and pub-sub events for a more immediate event dispatching.

Types

Network

name, chainId

Networkish

Can be a Network object, network name, chainId

FeeData

gasPrice //for legacy transactions - not EIP-1559.
maxFeePerGas, .maxPriorityFeePerGas

Block

hash, number, timestamp

Log

address //The address of the contract that generated this log
data //The data included in this log
topics //The list of topics (indexed properties) for this log

Transactions

A transaction request (**TransactionRequest**) describes a transaction that is to be sent to the network:

to, from, nonce, data, value (in wei)
gasLimit //The maximum amount of gas this transaction is permitted to use
gasPrice //The price (in wei) per unit of gas this transaction will pay
maxFeePerGas //The maximum price (in wei) per unit of gas - EIP1559
maxPriorityFeePerGas //tip for miners

A **TransactionResponse** includes all properties of a Transaction as well as several other

blockNumber, blockHash, timestamp, confirmations
wait //Resolves to the TransactionReceipt

If the transaction execution failed (i.e. the receipt status is 0), a CALL_EXCEPTION error will be rejected with the following properties: error.transaction, error.transactionHash, error.receipt

Transactions are replaced when the user uses an option in their client to send a new transaction from the same account with the original nonce. This is usually to speed up a transaction

TransactionReceipt

to, from, contractAddress, gasUsed
logsBloom //A bloom-filter, which includes all the addresses and topics
logs //All the logs emitted by this transaction
status //The status of a transaction is 1 is successful or 0 if it was reverted

Signers

A Signer is an abstraction of an Ethereum Account, which can be used to sign messages and transactions and send signed transactions. The available operations depend on the sub-class used. For example, a Signer from MetaMask can send transactions and sign messages but cannot sign a transaction.

Most common Signers:

Wallet - knows its private key
JsonRpcSigner - connected to a JsonRpcProvider

The Signer class is abstract and cannot be directly instantiated. Instead use one of the sub-classes, such as the Wallet or JsonRpcSigner.

```
signer.connect( provider ) => Signer
signer.getAddress( ) => Promise< string< Address > >
signer.getBalance( [ blockTag = "latest" ] ) => Promise< BigNumber >
signer.getTransactionCount( [ blockTag = "latest" ] ) => Promise< number >
signer.call( transactionRequest ) //account address used as the from field
signer.estimateGas( transactionRequest ) //account address used as the from field
```

Signing:

A signed message is prefixd with "\x19Ethereum Signed Message:\n" and the length of the message. A common case is to sign a hash. In this case, if the hash is a string, it must be converted to an array first, using the arrayify utility function

```
signer.signMessage( message ) => Promise< string< RawSignature > >

signer.signTransaction( transactionRequest ) => Promise< string< DataHexString > >

signer.sendTransaction( transactionRequest ) => Promise< TransactionResponse > //populates
transactionRequest with missing fields
```

Wallet

The Wallet class can sign transactions and messages using a private key.

```
//Create Wallet for PK and optionally connected to the provider.
new ethers.Wallet( privateKey [ , provider ] )

mnemonic = "announce room ..."
walletMnemonic = ethers.Wallet.fromMnemonic( mnemonic )

walletFromPrivateKey = new Wallet(walletMnemonic.privateKey)
```

```
await walletMnemonic.getAddress()

// Signing a message
await walletMnemonic.signMessage("Hello World")

// Signing a transaction - from and other fields are filled automatically
tx = {
  to: "0x8ba1f109551bD432803012645Ac136ddd64DBA72",
  value: utils.parseEther("1.0")
}

await walletMnemonic.signTransaction(tx)

// returns a new instance of the Wallet connected to a provider
wallet = walletMnemonic.connect(provider)

await wallet.getBalance();
await wallet.getTransactionCount();

//send Txn (send ETH)
await wallet.sendTransaction(tx)
```

Contract Interaction

Create contract instance:

```
new ethers.Contract( address , abi , signerOrProvider ) ⇒ Contract

//passing a Provider, creates a Contract with read-only access (calls)
contract.connect( providerOrSigner ) ⇒ Contract
```

Properties:

```
contract.address
contract.interface => Interface //ABI as an Interface.
contract.provider => Provider
contract.signer => Signer
```

Methods:

```
contract.deployed( ) ⇒ Promise< Contract >
```

Events:

//Return Events that match the event
contract.queryFilter(event [, fromBlock [, toBlock]) => Promise< Array< Event > >

contract.listeners(event) => Array< Listener >

Subscribe to event calling listener when the event occurs
contract.on(event , listener)
contract.once(event , listener)
contract.removeAllListeners([event])

Calling read-only methods:

//overrides.from, overrides.value, overrides.gasPrice...
//Result object will be returned with each parameter available
contract.functions.METHOD_NAME(...args [, overrides]) => Promise< Result >

Return values: For numbers, if the type is in the JavaScript safe range (i.e. less than 53 bits, such as an int24 or uint48) a normal JavaScript number is used. Otherwise a BigNumber is returned. For bytes (both fixed length and dynamic), a DataHexString is returned.

If the call reverts (or runs out of gas), a CALL_EXCEPTION will be thrown which will include:

error.address - the contract address
error.args - the arguments passed into the method
error.transaction - the transaction

Calling write methods:

contract.METHOD_NAME(...args [, overrides]) => Promise< TransactionResponse >

If the wait() method on the returned TransactionResponse is called, there will be additional properties on the receipt:

receipt.events - an array of the logs
receipt.events[n].args - the parsed arguments ...

contract.estimateGas.METHOD_NAME(...args [, overrides]) => Promise< BigNumber >

//Return UnsignedTransaction => represents the txn that needs to be signed and submitted
contract.populateTransaction.METHOD_NAME(...args [, overrides]) => Promise< UnsignedTx >

Event Filters

An event filter is made up of topics, which are values logged in a Bloom Filter, allowing efficient searching for entries which match a filter.

```
contract.filters.EVENT_NAME( ...args ) => Filter //Return a filter for EVENT_NAME
```

Only indexed event parameters may be filtered.

Meta-Class Methods (added at Runtime)

The methods available here depend on the ABI - eg:

```
myContract.decimals ; myContract.balanceOf(address)...

// Transfer 1.23 tokens to the ENS name "ricmoo.eth"
tx = await myContract.transfer("ricmoo.eth", parseUnits("1.23"));

// Wait for the transaction to be mined...
await tx.wait();

await myContract.estimateGas.transfer("ricmoo.eth", parseUnits("1.23"));
```

Meta-Class Filters (added at Runtime)

```
//Returns a new Filter to query or to subscribe/unsubscribe to events.
filterFrom = myContract.filters.Transfer(signer.address, null)
filterTo = myContract.filters.Transfer(null, signer.address);

// Search for transfers *from* me in the last 10 blocks
logsFrom = await myContract.queryFilter(filterFrom, -10, "latest");

// get args of event
logsFrom[0].args

// Listen to incoming events from signer:
myContract.on(filterFrom, (from, to, amount, event) => {
  // The `from` will always be the signer address
});

// Listen to all Transfer events:
myContract.on("Transfer", (from, to, amount, event) => {
  // ...
});
```

ContractFactory

A ContractFactory is an abstraction of a contract's bytecode and facilitates deploying a contract.

```
new ethers.ContractFactory( interface , bytecode [ , signer ] )

//pass constructor args
contractFactory.deploy( ...args [ , overrides ] ) ⇒ Promise< Contract >

//in Hardhat we can use:
const myContractFactory = await ethers.getContractFactory("myContractName")
const myContract = await myContractFactory.deploy("Deployment...")
await myContract.deployed()

//get the address
myContract.address

myContract.deployTransaction

// Wait until the transaction is mined
await myContract.deployTransaction.wait()
```

Utilities

AbiCoder

Most developers will never need to use this class directly, since the Interface class greatly simplifies these operations.

```
new ethers.utils.AbiCoder()

abiCoder.encode( types , values ) ⇒ string< DataHexString >
abiCoder.encode([ "uint", "string" ], [ 1234, "Hello World" ]);

abiCoder.decode( types , data ) ⇒ Result
data = "0x0000000000000000...";
abiCoder.decode([ "uint", "string" ], data);
```

Human-Readable ABI: A Human-Readable ABI is simple an array of strings, where each string is the Solidity signature.

```
const humanReadableAbi = [
  "function transferFrom(address from, address to, uint value)", ...  ]
```

Solidity JSON ABI: This is the ABI exported by Solidity compiler

Fragments

An ABI is a collection of Fragments, where each fragment specifies a Function, Error, Event or Constructor.

ethers.utils.Fragment.from(objectOrString) ⇒ Fragment

Properties:

fragment.name ⇒ string //name of event or function
fragment.type ⇒ string //function, event or constructor
fragment.inputs ⇒ Array< ParamType >

There are the following Fragment types: FunctionFragment, ErrorFragment, EventFragment, ConstructorFragment

ethers.utils.FunctionFragment.from(objectOrString) ⇒ FunctionFragment

ParamType:

paramType.name ⇒ string
paramType.type ⇒ string

Interface

The Interface Class abstracts the encoding and decoding required to interact with contracts.

```
//Create a new Interface from a JSON string or ABI object or Human-Readable Abi
new ethers.utils.Interface( abi )

const iface = new Interface([
  "constructor(string symbol, string name)",
  "function transferFrom(address from, address to, uint amount)", ... ]
```

Properties:

interface.fragments ⇒ Array< Fragment >
interface.events ⇒ Array< EventFragment >
interface.functions ⇒ Array< FunctionFragment >

Fragment Access

```
iface.getFunction("transferFrom(address, address, uint256)"); //returns FunctionFragment
iface.getError("AccountLocked(address, uint256)");
```

```
iface.getEvent("Transfer(address, address, uint256)");

iface.getSighash("balanceOf(address)");  // '0x70a08231'

iface.getEventTopic("Transfer(address, address, uint)"); // '0xddf252ad1be2c8...
```

Encoding Data

```
//Returns the encoded data, which can be used as the data for a transaction
interface.encodeFunctionData( fragment [ , values ] ) ⇒ string< DataHexString >

// Encoding data for the tx.data of a call or transaction
iface.encodeFunctionData("transferFrom", [
  "0x8ba1f109551bD432803012645Ac136ddd64DBA72",
  parseEther("1.0")
])

user = [
   "Richard Moore",
   "0x8ba1f109551bD432803012645Ac136ddd64DBA72"
];
iface.encodeFunctionData("addUser", [ user ]);
```

Decoding Data

```
//Returns the decoded values from transaction data for fragment
interface.decodeFunctionData( fragment , data ) ⇒ Result

// Decoding function data (the value of tx.data)
const txData = "0x23b872dd000000000000000000000000008ba1...";
iface.decodeFunctionData("transferFrom", txData);

//Returns the decoded values from the result of a call for fragment
interface.decodeFunctionResult( fragment , data ) ⇒ Result

// Decoding result data (e.g. from an eth_call)
resultData = "0x0000000000000000de0b6b3a7640000...";
iface.decodeFunctionResult("balanceOf", resultData)
```

Parsing

```
const data = "0xf7c3865a0000000000000000...";
const value = parseEther("1.0");

const topics = [
  "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
```

```
  "0x0000000000000000000000008ba1f109551bd432803012645ac136ddd64dba72",
  "0x000000000000000000000000ab7c8803962c0f2f5bbbe3fa8bf41cd82aa1923c"
];

iface.parseError(data);
iface.parseTransaction({ data, value });
iface.parseLog({ data, topics });
```

Addresses

An Address is a DataHexString of 20 bytes.

```
ethers.utils.getAddress( address ) ⇒ string< Address >source //Returns Checksum Address
ethers.utils.isAddress( address ) ⇒ boolean
ethers.utils.computeAddress( publicOrPrivateKey ) ⇒ string< Address >

ethers.utils.recoverAddress( digest , signature ) ⇒ string< Address >

const digest = "0x7c5ea36004851c764c44143b1dcb59679b11cf3d480715331";
recoverAddress(digest, {
  r: "0x528459e4aec8934dc2ee94c4f3265cf6ce00d47cf42bb106afda3642c72e25eb",
  s: "0x42544137118256121502784e5a6425e6183ca964421ecd577db6c66ba9bccdcf",
  v: 27
});

// Using a flat Signature
const signature = "0x528459e4aec8934dc2ee94c4f3265cf6ce00d47cba9bccdcf1b";
recoverAddress(digest, signature);
```

BigNumber

Many operations in Ethereum operate on numbers which are outside the range of safe values to use in JavaScript. A BigNumber is an object which safely allows mathematical operations on numbers of any magnitude. Most operations which need to return a value will return a BigNumber

The constructor of BigNumber cannot be called directly => use BigNumber.from.

```
//aBigNumberish: can be string, BigNumber, BigInt (JS), number (save JS), BytesLike
ethers.BigNumber.from( aBigNumberish ) ⇒ BigNumber

BigNumber.from("42")
BigNumber.from("0x2a")
BigNumber.from(42)
```

Math operations:

BigNumber.add(otherValue) ⇒ BigNumber
sub, mul, div, mod, pow, abs

Comparison:

BigNumber.eq(otherValue) ⇒ boolean
lt, lte, gt, gte

Conversion:

BigNumber.toBigInt( ) ⇒ bigint
toNumber, toString, toHexString

The functions parseEther(etherString) and formatEther(wei) can be used to convert between string representations

Byte Manipulation

Types: Bytes, DataHexString, HexString, Signature (r,s,v...), Raw Signature

//Converts DataHexStringOrArrayish to a Uint8Array
ethers.utils.arrayify(DataHexStringOrArrayish [, options]) ⇒ Uint8Array
arrayify("0x1234") // Uint8Array [18, 52]

//Converts aBigNumberish to a HexString
ethers.utils.hexValue(aBigNumberish) ⇒ string< HexString >
hexValue(1) // '0x1'
hexValue([1, 2]) // '0x102'

Constants

ethers.constants.AddressZero ⇒ string< Address > // 20 bytes
ethers.constants.Zero ⇒ BigNumber
ethers.constants.One ⇒ BigNumber
ethers.constants.WeiPerEther ⇒ BigNumber
ethers.constants.MaxUint256 ⇒ BigNumber

Display Logic and Input

A Wallet may specify the balance in ether, and gas prices in gwei for the User Interface, but when sending a transaction, both must be specified in wei. The parseUnits will parse a string representing ether, such as 1.1 into a BigNumber in wei. The formatUnits will format a BigNumberish into a string, which is useful when displaying a balance.

//Returns a string of value formatted with unit digits or to the unit specified
ethers.utils.formatUnits(value [, unit = "ether"]) ⇒ string

const oneGwei = BigNumber.from("1000000000");
formatUnits(oneGwei, 0); // '1000000000'
formatUnits(oneGwei, "gwei"); // '1.0'
formatUnits(oneGwei, 9); // '1.0'

//The equivalent to calling formatUnits(value, "ether")
ethers.utils.formatEther(value) ⇒ string

ethers.utils.parseUnits(value [, unit = "ether"]) ⇒ BigNumber
parseUnits("121.0", "gwei"); // { BigNumber: "121000000000" }
parseUnits("121.0", 9); // { BigNumber: "121000000000" }

//The equivalent to calling parseUnits(value, "ether")
ethers.utils.parseEther(value) ⇒ BigNumber

Encoding Utilities

ethers.utils.base64.decode(textData) ⇒ Uint8Array
base64.decode("EjQ="); // Uint8Array [18, 52]

ethers.utils.base64.encode(aBytesLike) ⇒ string
base64.encode("0x1234"); // 'EjQ='

Fixed Number

A FixedNumber is a fixed-width (in bits) number with an internal base-10 divisor, which allows it to represent a decimal fractional component.

FixedNumber.from(value [, format = "fixed"]) ⇒ FixedNumber
FixedNumber.fromValue(value [, decimals = 0 [, format = "fixed"]]) ⇒ FixedNumber

Methods:

fixednumber.addUnsafe(otherValue) ⇒ FixedNumber
fixednumber.subUnsafe(otherValue) ⇒ FixedNumber
fixednumber.mulUnsafe(otherValue) ⇒ FixedNumber
fixednumber.divUnsafe(otherValue) ⇒ FixedNumber
fixednumber.round([decimals = 0]) ⇒ FixedNumber

Hashing Algorithms

//The Ethereum Identity function computes the KECCAK256 hash of the text bytes.
ethers.utils.id(text) ⇒ string< DataHexString< 32 > >

```
ethers.utils.keccak256( aBytesLike ) ⇒ string< DataHexString< 32 > >

utils.keccak256([ 0x12, 0x34 ]) // '0x56570de...'
utils.keccak256("0x1234") // '0x56570de...'

// If needed, convert strings to bytes first:
utils.keccak256(utils.toUtf8Bytes("hello world"))

// Or equivalently use the identity function:
utils.id("hello world")

//Computes the EIP-191 personal message digest of message. Personal messages
//are converted to UTF-8 bytes and prefixed with \x19Ethereum Signed Message:
//and the length of message
ethers.utils.hashMessage( message ) ⇒ string< DataHexString< 32 > >

utils.hashMessage("Hello World") // '0xa1de988600a4...'

// Hashing binary data (also "Hello World", but as bytes) => '0xa1de988600a4...'
utils.hashMessage( [ 72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100 ])
```

Solidity Hashing Algorithms

When using the Solidity abi.encodePacked(...) function, a non-standard tightly packed version of encoding is used. These functions implement the tightly packing algorithm.

```
ethers.utils.solidityKeccak256( types , values ) ⇒ string< DataHexString< 32 > >
utils.solidityKeccak256([ "int16", "uint48" ], [ -1, 12 ])
```

Strings

```
//Returns the decoded string represented by the Bytes32 encoded data.
ethers.utils.parseBytes32String( aBytesLike ) ⇒ string

//Returns a bytes32 string representation of text.
ethers.utils.formatBytes32String( text ) ⇒ string< DataHexString< 32 > >

ethers.utils.toUtf8Bytes( text [ , form = current ] ) ⇒ Uint8Array
ethers.utils.toUtf8String( aBytesLike [ , onError = error ] ) ⇒ string
```

Transactions

A generic object to represent a transaction.

Properties:

hash, to, from, nonce, data, value, gasLimit , maxFeePerGas, maxPriorityFeePerGas, v, r, s

//Parses the transaction properties from a serialized transaction.
ethers.utils.parseTransaction(aBytesLike) ⇒ Transaction
ethers.utils.serializeTransaction(tx [, signature]) ⇒ string< DataHexString >

Compute the raw transaction:

```
function getRawTransaction(tx) {

  function addKey(accum, key) {
    if (tx[key]) { accum[key] = tx[key]; }
    return accum;
  }

  // Extract the relevant parts of the transaction and signature
  const txFields = "accessList chainId data gasPrice gasLimit maxFeePerGas
maxPriorityFeePerGas nonce to type value".split(" ");
  const sigFields = "v r s".split(" ");

  // Seriailze the signed transaction
  const raw = utils.serializeTransaction(txFields.reduce(addKey, { }),
sigFields.reduce(addKey, { }));

  // Double check things went well
  if (utils.keccak256(raw) !== tx.hash) { throw new Error("serializing failed!"); }

  return raw;
}
```