gnosis.js Documentation

Gnosis

May 08, 2018

Contents

CHAPTER 1

Developer Guide

In order to follow this guide, you will need to be comfortable working in your OS's shell, writing JavaScript, and working with npm. A working knowledge of Ethereum, Solidity, and Truffle would greatly ease the use of this library, but is not strictly necessary for getting started. The usage of a VCS such as Git is also encouraged, but is not explained here.

The Gnosis JavaScript library should work for any OS. It has the following system requirements:

- Node.js (versions >= 7 should work)
- NPM (should be installed with Node.js, and versions >= 5 should work)
- Git (optional)

1.1 Installation

1.1.1 Setting up a project using npm

1. Create a project directory, open a terminal or command line, and change directory into the project directory.
2. Use `npm init` to set up your project.
3. Install `gnosis.js` into your project as a dependency using:

   ```
   npm install --save @gnosis.pm/gnosisjs
   ```

   Be sure to issue this command with this exact spelling.

   This command installs the Gnosis JavaScript library and its dependencies into the `node_modules` directory. The `@gnosis.pm/gnosisjs` package contains the following:

   - ES6 source of the library in `src` which can also be found on the repository
   - Compiled versions of the modules which can be run on Node.js in the `dist` directory
   - Webpacked standalone `gnosis[.min].js` files ready for use by web clients in the `dist` directory

- API documentation in the `docs` directory

Notice that the library refers to the `dist/index` module as the `package.json` main. This is because even though Node.js does support many new JavaScript features, it natively does not support the use of ES6 imports yet (see this issue), so the modules are transpiled with Babel for Node interoperability.

In the project directory, you can experiment with the Gnosis API by opening up a `node` shell and importing the library like so:

```
const Gnosis = require('@gnosis.pm/gnosisjs')
```

This will import the transpiled library through the `dist/index` entry point, which exports the Gnosis class.

If you are playing around with gnosis.js directly in the project folder, you can import it from dist

```
const Gnosis = require('.')
```

1.1.2 Browser use

The `gnosis.js` file and its minified version `gnosis.min.js` are self-contained and can be used directly in a webpage. For example, you may copy `gnosis.min.js` into a folder or onto your server, and in an HTML page, use the following code to import the library:

```
<script src="gnosis.min.js"></script>
<script>
// Gnosis should be available as a global after the above script import, so this␣
↪subsequent script tag can make use of the API.
</script>
```

After opening the page, the browser console can also be used to experiment with the API.

1.1.3 Integration with webpack projects (advanced)

The ES6 source can also be used directly with webpack projects. Please refer to the Babel transpilation settings in `.babelrc` and the webpack configuration in `webpack.config.js` to see what may be involved.

1.1.4 Setting up an Ethereum JSON RPC

After setting up the Gnosis.js library, you will still need a connection to an Ethereum JSON RPC provider. Without this connection, the following error occurs when trying to use the API to perform actions with the smart contracts:

```
Error: Invalid JSON RPC response: ""
```

Gnosis.js refers to Truffle contract build artifacts found in `node_modules/@gnosis.pm/gnosis-core-contracts/build/contracts/`, which contain a registry of where key contracts are deployed given a network ID. By default Gnosis contract suite is already deployed on the Ropsten, Kovan, and Rinkeby testnets.

Ganache-cli and private chain providers

Ganache-cli is a JSON RPC provider which is designed to ease developing Ethereum dapps. It can be used in tandem with Gnosis.js as well, but its use requires some setup. Since Ganache-cli randomly generates a network ID and begins the Ethereum VM in a blank state, the contract suite would need to be deployed, and the deployed contract addresses

recorded in the build artifacts before use with Ganache-cli. This can be done by running the migration script in the core contracts package directory.

```
(cd node_modules/\@gnosis.pm/gnosis-core-contracts/ && truffle migrate)
```

This will deploy the contracts onto the chain and will record the deployed addresses in the contract build artifacts. This will make the API available to Gnosis.js applications which use the transpiled *modules* in `dist` (typically Node.js apps), as these modules refer directly to the build artifacts in the `@gnosis.pm/gnosis-core-contracts` package. However, for browser applications which use the standalone library file `gnosis[.min].js`, that file has to be rebuilt to incorporate the new deployment addresses info.

MetaMask

MetaMask is a Chrome browser plugin which injects an instrumented instance of Web3.js into the page. It comes preloaded with connections to the Ethereum mainnet as well as the Ropsten, Kovan, and Rinkeby testnets through Infura. Gnosis.js works out-of-the-box with MetaMask configured to connect to these testnets. Make sure your web page is being served over HTTP/HTTPS and uses the standalone library file.

1.2 API Overview

The Gnosis.js library is encapsulated inside of the Gnosis class. In order for it to function, it must be connected to an Ethereum network through a Web3.js interface. It also uses IPFS for publishing and retrieving event data, and so it will also have to be connected to an IPFS node. Configuration of these connections can be done with a call to the asynchronous factory function Gnosis.create. For example, the following code will store an instance of the Gnosis.js library into the variable `gnosis`:

```
let gnosis

Gnosis.create().then(result => {
    gnosis = result
    // gnosis is available here and may be used
})

// note that gnosis is NOT guaranteed to be initialized outside the callback scope␣
↪here
```

Because of the library's dependence on remote service providers and the necessity to wait for transactions to complete on the blockchain, the majority of the methods in the API are asynchronous and return thenables in the form of Promises.

Gnosis.js also relies on Truffle contract abstractions. In fact, much of the underlying core contract functionality can be accessed in Gnosis.js as one of these abstractions. Since the Truffle contract wrapper has to perform asynchronous actions such as wait on the result of a remote request to an Ethereum RPC node, it also uses thenables. For example, here is how to use the on-chain Gnosis Math library exposed at Gnosis.contracts to print the approximate natural log of a number:

```
const ONE = Math.pow(2, 64)
Gnosis.create()
    .then(gnosis => gnosis.contracts.Math.deployed())
    .then(math => math.ln(3 * ONE))
    .then(result => console.log('Math.ln(3) =', result.valueOf() / ONE))
```

Although it is not strictly necessary, usage of `async/await` syntax is encouraged for simplifying the use of thenable programming, especially in complex flow scenarios. To increase the readability of code examples from this

point forward, this guide will assume `async/await` is available and snippets execute in the context of an `async function`. With those assumptions, the previous example can be expressed like so:

```
const ONE = Math.pow(2, 64)
const gnosis = await Gnosis.create()
const math = await gnosis.contracts.Math.deployed()
console.log('Math.ln(3) =', (await math.ln(3 * ONE)).valueOf() / ONE)
```

Gnosis.js also exposes a number of convenience methods wrapping contract operations such as Gnosis.createCentralizedOracle and Gnosis.createScalarEvent.

1.3 Events, Oracles and Markets

1.3.1 Questions About the Future, Oracles, and Trust

A prediction predicts the outcome of a future event. For example, the event might be "the U.S. presidential election of 2016." There may be predictions associated with each of the possible outcomes, but this event only had one of these outcome. Events like these with a discrete set of outcomes are considered to be categorical events. They may be phrased as a question with a choice of answers, e.g.:

Who will win the U.S. presidential election of 2016?

- Clinton
- Trump
- Other

To ask this question with a prediction market on Gnosis, you must first upload the event description onto IPFS via Gnosis.publishEventDescription. This will asynchronously provide you with a hash value which can be used to locate the file on IPFS:

```
let gnosis, ipfsHash
async function createDescription () {
    gnosis = await Gnosis.create()
    ipfsHash = await gnosis.publishEventDescription({
        title: 'Who will win the U.S. presidential election of 2016?',
        description: 'Every four years, the citizens of the United States vote for␣
↪their next president...',
        resolutionDate: '2016-11-08T23:00:00-05:00',
        outcomes: ['Clinton', 'Trump', 'Other'],
    })
    // now the event description has been uploaded to ipfsHash and can be used
    console.assert(
        (await gnosis.loadEventDescription(ipfsHash)).title ===
        'Who will win the U.S. presidential election of 2016?',
    )
    console.info(`Ipfs hash: https://ipfs.infura.io/api/v0/cat?stream-channels=true&
↪arg=${ipfsHash}`)
}
createDescription()
```

Of course, future events will come to pass, and once they do, the outcome should be determinable. Oracles report on the outcome of events. The simplest oracle contract provided by Gnosis is a `CentralizedOracle`, and it is controlled by a single entity: the `owner` of the contract, which is a single Ethereum address, and which will from this point forward in this guide be referred to as the centralized oracle itself.

To create a centralized oracle, use Gnosis.createCentralizedOracle:

```
// After obtaining an instance of Gnosis (api-reference.html#Gnosis) as "gnosis" and
↪"ipfsHash" from Gnosis.publishEventDescription (api-reference.html
↪#publishEventDescription)
let oracle
async function createOracle() {
    oracle = await gnosis.createCentralizedOracle(ipfsHash)
    console.info(`Oracle created with address ${oracle.address}`)
}
createOracle()
```

After `createCentralizedOracle` finishes, the owner of the CentralizedOracle contract instance created will be the message sender, or the default account for all transactions in the Gnosis instance (which is normally set to the first account exposed by the Web3 provider).

By no means is the CentralizedOracle the only possible oracle design which can be used with Gnosis. Any oracle which implements the `Oracle` contract interface may be used.

1.3.2 Events and Collateral

Once an oracle is created, an event contract may defer to the oracle's judgment. The `CategoricalEvent` and `ScalarEvent` contracts represent an event. They also mint sets of outcome tokens corresponding to a collateral of an ERC20-compliant token. Once the relied-on oracle reports an outcome to the event, the outcome token corresponding to the reported outcome may be exchanged for the original collateral token.

Note that ether is *not* an ERC20-compliant token at the moment of this writing. It may be converted into an ERC20-compliant variant with an adaptor contract like EtherToken though. There is a deployed instance of EtherToken available in the API as Gnosis.etherToken.

In order to create a categorical event contract instance backed by an specific `oracle`, use Gnosis.createCategoricalEvent. For example, a categorical event with three outcomes like the earlier example can be made like this:

```
let event
async function createCategoricalEvent() {
    event = await gnosis.createCategoricalEvent({
        collateralToken: gnosis.etherToken,
        oracle,
        // Note the outcomeCount must match the length of the outcomes array␣
↪published on IPFS
        outcomeCount: 3,
    })
    console.info(`Categorical event created with address ${event.address}`)
}
createCategoricalEvent()
```

Note that EtherToken is traded with this particular event instance.

> If you are using the Rinkeby network, you can check that your event has been indexed by GnosisDB with the following URL:
>
> ```
> https://gnosisdb.rinkeby.gnosis.pm/api/events/EVENT_ADDRESS
> ```
>
> Be sure to substitute in the actual event address for `EVENT_ADDRESS`

When an event has been created, users can convert their collateral into sets of outcome tokens. For example, suppose a user buys 4 ETH worth of outcome tokens from `event`:

```
async function buyAllOutcomes() {
    const txResults = await Promise.all([
        [gnosis.etherToken.constructor, await gnosis.etherToken.deposit.
↪sendTransaction({ value: depositValue })],
        [gnosis.etherToken.constructor, await gnosis.etherToken.approve.
↪sendTransaction(event.address, depositValue)],
        [event.constructor, await event.buyAllOutcomes.sendTransaction(depositValue)],
    ].map(([contract, txHash]) => contract.syncTransaction(txHash)))

    // Make sure everything worked
    const expectedEvents = [
        'Deposit',
        'Approval',
        'OutcomeTokenSetIssuance',
    ]
    txResults.forEach((txResult, i) => {
        Gnosis.requireEventFromTXResult(txResult, expectedEvents[i])
    })
}
buyAllOutcomes()
```

The computation of `txResults` may appear to be fairly complex in the previous example, so here is a breakdown of that computation and why it was done that way.

At first, you may think of the following code:

```
const txResults = [
    await gnosis.etherToken.deposit({ value: depositValue }),
    await gnosis.etherToken.approve(event.address, depositValue),
    await event.buyAllOutcomes(depositValue),
]
```

This code would actually be just fine, and would work in place of the above, but would provide a flawed user experience. Users would have to wait for the `deposit` and `approve` transactions to *mine* before the next transaction is even sent, and on the Ethereum mainnet, this could take 15-30s between each transaction!

Then, you may propose:

```
const txResults = await Promise.all([
    gnosis.etherToken.deposit({ value: depositValue }),
    gnosis.etherToken.approve(event.address, depositValue),
    event.buyAllOutcomes(depositValue),
])
```

Indeed, all the transactions are sent to the provider and mined simultaneously, but there is no order to them. In practice, this usually means it is up to the user to sign the transactions in order: this is also a flawed user experience, as there is only one order here which makes any sense.

This is why we send in the transactions like so:

```
const txHash = await gnosis.etherToken.deposit.sendTransaction({ value:␣
↪depositValue })
```

The `sendTransaction` variant of these Truffle methods only wait until the transaction hash is determined. Since the transaction hash is partially derived from the user's account nonce, we can ensure that the transactions are processed in order. Then, we can wait for all of the transactions we sent in to mine:

```
const txResults = await Promise.all([gnosis.etherToken.constructor.
↪syncTransaction(depositTransactionHash), ...])
```

You may notice that the constructor of the contract instance was used to call a method `syncTransaction`. That constructor is the same as the contract type abstraction, that is: `gnosis.etherToken.constructor === gnosis.contracts.EtherToken`. This is not an official method of `truffle-contract` yet! For more information, see this pull request.

After executing a `buyAllOutcomes` transaction as above, the user would then have `4e18` units of each `OutcomeToken`:

```
async function checkBalances() {
    const { Token } = gnosis.contracts
    const outcomeCount = (await event.getOutcomeCount()).valueOf()

    for(let i = 0; i < outcomeCount; i++) {
        const outcomeToken = await Token.at(await event.outcomeTokens(i))
        console.log('Have', (await outcomeToken.balanceOf(gnosis.defaultAccount)).
↪valueOf(), 'units of outcome', i)
    }
}
checkBalances()
```

Finally, if you are the centralized oracle for an `event` contract which refers to the 2016 U.S. presidential election as set up above, you can report the outcome of the event as "Trump" and allow stakeholders to settle their claims with Gnosis.resolveEvent:

```
async function resolve() {
    await gnosis.resolveEvent({ event, outcome: 1 })
}
resolve()
```

Note that you must pass in the 0-based index of the outcome corresponding to the event description published on IPFS ("Trump" has index 1 in the example `['Clinton', 'Trump', 'Other']`),

If you are a stakeholder in this `event` contract instance, you can redeem your winnings with `CategoricalEvent.redeemWinnings`:

```
async function redeem() {
    Gnosis.requireEventFromTXResult(await event.redeemWinnings(), 'WinningsRedemption
↪')
}
redeem()
```

1.3.3 Markets and Automated Market Makers

Suppose that Alice believed Clinton would win the 2016 U.S. election, but Bob believed Trump would win that election. With the machinery we've developed thus far, both Alice and Bob would have to buy outcome tokens and then trade each other based on their beliefs. Alice would give Trump tokens to Bob in exchange for Clinton tokens. When the oracle reports that the outcome of the election was Trump, the Trump tokens held by Bob can be exchanged for the collateral used to back those tokens.

However, it may be difficult to coordinate the trade. In order to create liquidity, an automated market maker may be used to operate an on-chain market. These markets also aggregate information from participants about their beliefs about the likeliness of outcomes.

Gnosis contracts contain market and market maker contract interfaces, a standard market implementation, and an implementation of the logarithmic market scoring rule (LMSR), an automated market maker. This can be leveraged with the Gnosis.createMarket method. For example, given an `event`, you can create a `StandardMarket` operated by the LMSR market maker with the following:

```
let market
async function createMarket() {
    market = await gnosis.createMarket({
        event,
        marketMaker: gnosis.lmsrMarketMaker,
        fee: 50000 // signifies a 5% fee on transactions
            // see docs at Gnosis.createMarket (api-reference.html#createMarket) for more info
    })
    console.info(`Market created with address ${market.address}`)
}
createMarket()
```

Once a `market` has been created, it needs to be funded with the collateral token in order for it to provide liquidity. The market creator funds the market according to the maximum loss acceptable to them, which is possible since LMSR guarantees a bounded loss:

```
async function fund() {
    // Fund the market with 4 ETH
    const txResults = await Promise.all([
        [gnosis.etherToken.constructor, await gnosis.etherToken.deposit.sendTransaction({ value: 4e18 })],
        [gnosis.etherToken.constructor, await gnosis.etherToken.approve.sendTransaction(market.address, 4e18)],
        [market.constructor, await market.fund.sendTransaction(4e18)],
    ].map(([contract, txHash]) => contract.syncTransaction(txHash)))

    const expectedEvents = [
        'Deposit',
        'Approval',
        'MarketFunding',
    ]
    txResults.forEach((txResult, i) => {
        Gnosis.requireEventFromTXResult(txResult, expectedEvents[i])
    })
}
fund()
```

Furthermore, the outcome tokens sold by the market are guaranteed to be backed by collateral because the ultimate source of these outcome tokens are from the event contract, which only allow buying collateral-backed sets of outcome tokens.

Let's suppose there is a `market` on the 2016 presidential election as indicated above, and that you are wondering if "Other" outcome tokens (which have index 2) are worth it at its price point. You can estimate how much it would cost to buy `1e18` units of those outcome tokens with `LMSRMarketMaker.calcCost`:

```
async function calcCost() {
    const cost = await gnosis.lmsrMarketMaker.calcCost(market2.address, 2, 1e18)
    console.info(`Buy 1 Outcome Token with index 2 costs ${cost.valueOf()/1e18} ETH tokens`)
}
calcCost()
```

Let's say now that you've decided that these outcome tokens are worth it. Gnosis.js contains convenience functions for buying and selling outcome tokens from a market backed by an LMSR market maker. They are Gnosis.buyOutcomeTokens and Gnosis.sellOutcomeTokens. To buy these outcome tokens, you can use the following code:

```
async function buyOutcomeTokens() {
    await gnosis.buyOutcomeTokens({
        market,
        outcomeTokenIndex: 2,
        outcomeTokenCount: 1e18,
    })
    console.info('Bought 1 Outcome Token of Outcome with index 2')
}
buyOutcomeTokens()
```

Similarly, you can see how much these outcome tokens are worth to the `market` with `LMSRMarketMaker.calcProfit`:

```
async function calcProfit() {
    const profit = await gnosis.lmsrMarketMaker.calcProfit(market.address, 2, 1e18)
    console.info(`Sell 1 Outcome Token with index 2 gives ${profit.valueOf()/1e18}␣
↪ETH tokens of profit`)
}
calcProfit()
```

If you want to sell the outcome tokens you have bought, you can do the following:

```
async function sellOutcomeTokens() {
    await gnosis.sellOutcomeTokens({
        market,
        outcomeTokenIndex: 2,
        outcomeTokenCount: 1e18,
    })
}
sellOutcomeTokens()
```

Oftentimes prediction markets aggregate predictions into more accurate predictions. Because of this, without a fee, the investor can expect to take a loss on their investments. However, too high of a fee would discourage participation in the market. Discerning the best fee factor for markets is outside the scope of this document.

Finally, if you are the creator of a `StandardMarket`, you can close the market and obtain all of its outcome token holdings with `StandardMarket.close` and/or withdraw the trading fees paid with `StandardMarket.withdrawFees`:

```
async function closeAndWithdraw() {
    Gnosis.requireEventFromTXResult(await market.close(), 'MarketClose')
    Gnosis.requireEventFromTXResult(await market.withdrawFees(), 'MarketFeeWithdrawal
↪')
}
closeAndWithdraw()
```

1.3.4 Events with Scalar Outcomes

The discussion up to this point has been about an instance of an event with categorical outcomes. However, some events may be better expressed as an event with a scalar outcome. For example, you can ask the following question using Gnosis.createScalarEvent:

```
const lowerBound = '80'
const upperBound = '100'

let ipfsHash, oracle, event

async function createScalarEvent() {
    ipfsHash = await gnosis.publishEventDescription({
        title: 'What will be the annual global land and ocean temperature anomaly for␣
↪2017?',
        description: 'The anomaly is with respect to the average temperature for the␣
↪20th century and is as reported by the National Centers for Environmental Services..
↪.',
        resolutionDate: '2017-01-01T00:00:00+00:00',
        lowerBound,
        upperBound,
        decimals: 2,
        unit: '°C',
    })

    console.info(`Ipfs hash: https://ipfs.infura.io/api/v0/cat?stream-channels=true&
↪arg=${ipfsHash}`)

    oracle = await gnosis.createCentralizedOracle(ipfsHash)

    console.info(`Oracle created with address ${oracle.address}`)

    event = await gnosis.createScalarEvent({
        collateralToken: gnosis.etherToken,
        oracle,
        // Note that these bounds should match the values published on IPFS
        lowerBound,
        upperBound,
    })

    console.info(`Event created with address ${event.address}`)
}
createScalarEvent()
```

This sets up an event with a lower bound of 0.80°C and an upper bound of 1.00°C. Note that the values are passed in as whole integers and adjusted to the right order of magnitude according to the `decimals` property of the event description.

There are two outcome tokens associated with this event: a short token for the lower bound and a long token for the upper bound. The short tokens associated with the lower bound have index 0, as opposed to the long tokens associated with the upper bound which have index 1. In other words, other than their value at resolution, they have the same mechanics as a categorical event with two outcomes. For example, a `market` may be created for this event in the same way, and outcome tokens traded on that market may also be traded in the same way.

Now let's say that the NCES reports that the average global temperature anomaly for 2017 is 0.89°C. If you are the centralized oracle for this event as above, you can report this result to the chain like so:

```
async function resolve() {
    await gnosis.resolveEvent({ event, outcome: '89' })
}
```

This will value each unit of the short outcome at \(1 - {0.89 - 0.80 \over 1.00 - 0.80} = 0.55\) units of the collateral, and the long outcome at \(0.45\) units of the collateral. Thus, if you held 50 units of the short outcome and 100 units of the long outcome, `ScalarEvent.redeemWinnings` would net you \(\lfloor 50 \times 0.55 + 100 \times 0.45

\rfloor = 72\) units of collateral. Hopefully you'll have paid less than that for those outcomes.

1.4 LMSR Primer

The Gnosis.js implementation of the logarithmic market scoring rule mostly follows the original specification. It is based on the following cost function:

$$ C(\vec{q}) = b \log \left(\sum_i \exp \left({ q_i \over b } \right) \right) $$

where

- \(\vec{q}\) is a vector of *net* quantities of outcome tokens *sold*. What this means is that although the market selling outcome tokens increases the net quantity sold, the market *buying* outcome tokens *decreases* the net quantity sold.
- \(b\) is a liquidity parameter which controls the bounded loss of the LMSR. That bounded loss for the market maker means that the liquidity parameter can be expressed in terms of the number of outcomes and the funding required to guarantee all outcomes sold by the market maker can be backed by collateral (this will be derived later).
- \(\log\) and \(\exp\) are the natural logarithm and exponential functions respectively

The cost function is used to determine the cost of a transaction in the following way: suppose \(\vec{q_1}\) is the state of net quantities sold before the transaction and \(\vec{q_2}\) is this state afterwards. Then the cost of the transaction \(\nu\) is

$$ \nu = C(\vec{q_2}) - C(\vec{q_1}) $$

For example, suppose there is a LMSR-operated market with a \(b\) of 5 and two outcomes. If this market has bought 10 tokens for outcome A and sold 4 tokens for outcome B, it would have a cost level of:

$$ C \begin{pmatrix} -10 \\ 4 \end{pmatrix} = 5 \log \left(\exp(-10/5) + \exp(4/5) \right) \approx 4.295 $$

Buying 5 tokens for outcome A (or having the market sell you those tokens) would change the cost level to:

$$ C \begin{pmatrix} -10 + 5 \\ 4 \end{pmatrix} = 5 \log \left(\exp(-5/5) + \exp(4/5) \right) \approx 4.765 $$

So the cost of buying 5 tokens for outcome A from this market is:

$$ \nu = C \begin{pmatrix} -5 \\ 4 \end{pmatrix} - C \begin{pmatrix} -10 \\ 4 \end{pmatrix} \approx 4.765 - 4.295 = 0.470 $$

Similarly, selling 2 tokens for outcome B (or having the market buy those tokens from you) would yield a cost of:

$$ \nu = C \begin{pmatrix} -10 \\ 2 \end{pmatrix} - C \begin{pmatrix} -10 \\ 4 \end{pmatrix} \approx -1.861 $$

That is to say, the market will buy 2 tokens of outcome B for 1.861 units of collateral.

1.4.1 Bounded Loss from the \(b\) Parameter

Here is the worst scenario for the market maker: everybody but the market maker already knows which one of the \(n\) outcomes will occur. Without loss of generality, let the answer be the first outcome token. Everybody buys outcome one tokens from the market maker while selling off every other worthless outcome token they hold. The cost function for the market maker goes from

$$ C \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix} = b \log n $$

to

$$ C \begin{pmatrix} q_1 \\ -\infty \\ -\infty \\ \vdots \end{pmatrix} = b \log \left(\exp \left({q_1 \over b} \right) \right) = q_1 $$

The market sells \(q_1\) shares of outcome one and buys shares for every other outcome until those outcome tokens become worthless to the market maker. This costs the participants \((q_1 - b \log n)\) in collateral, and thus, when the participants gain \(q_1\) from redeeming their winnings, this nets the participants \((b \log n)\) in collateral. This gain for the participant is equal to the market's loss.

Thus, in order to guarantee that a market can operate with a liquidity parameter of \(b\), it must be funded with \((F = b \log n)\) of collateral. Another way to look at this is that the market's funding determines its \(b\) parameter:

$$ b = {F \over \log n} $$

In the Gnosis implementation, the LMSR market maker contract is provided with the `funding` \(F\) through inspection of the `market`, and \(b\) is derived accordingly.

1.4.2 Marginal Price of Outcome Tokens

Because the cost function is nonlinear, there isn't a price for outcome tokens which scales with the quantity being purchased. However, the cost function *is* differentiable, so a marginal price can be quoted for infinitesimal quantities of outcome tokens:

$$ P_i = {\partial C(\vec{q}) \over \partial q_i} = \frac{\exp(q_i / b)}{\sum_k \exp(q_k / b)} $$

In the context of prediction markets, this marginal price can also be interpreted as the market's estimation of the odds of that outcome occurring.

1.4.3 LMSR Calculation Functions

The functions Gnosis.calcLMSRCost and Gnosis.calcLMSRProfit estimate the cost of buying outcome tokens and the profit from selling outcome tokens respectively. The Gnosis.calcLMSROutcomeTokenCount estimates the quantity of an outcome token which can be bought given an amount of collateral and serves as a sort of "inverse calculation" to Gnosis.calcLMSRCost. Finally, Gnosis.calcLMSRMarginalPrice can be used to get the marginal price of an outcome token.

CHAPTER 2

API Reference

2.1 Classes

class Gnosis(*opts*)

Represents the gnosis.js API

Warning: Do not use constructor directly. Some asynchronous initialization will not be handled. Instead, use Gnosis.create.

Gnosis.**contracts**

A collection of Truffle contract abstractions for the following Gnosis contracts:

- Math (https://gnosis.github.io/gnosis-contracts/docs/Math)
- Event (https://gnosis.github.io/gnosis-contracts/docs/Event)
- CategoricalEvent (https://gnosis.github.io/gnosis-contracts/docs/CategoricalEvent)
- ScalarEvent (https://gnosis.github.io/gnosis-contracts/docs/ScalarEvent)
- EventFactory (https://gnosis.github.io/gnosis-contracts/docs/EventFactory)
- Token (https://gnosis.github.io/gnosis-contracts/docs/Token)
- HumanFriendlyToken (https://gnosis.github.io/gnosis-contracts/docs/HumanFriendlyToken)
- EtherToken (https://gnosis.github.io/gnosis-contracts/docs/EtherToken)
- CentralizedOracle (https://gnosis.github.io/gnosis-contracts/docs/CentralizedOracle)
- CentralizedOracleFactory (https://gnosis.github.io/gnosis-contracts/docs/CentralizedOracleFactory)
- UltimateOracle (https://gnosis.github.io/gnosis-contracts/docs/UltimateOracle)
- UltimateOracleFactory (https://gnosis.github.io/gnosis-contracts/docs/UltimateOracleFactory)
- LMSRMarketMaker (https://gnosis.github.io/gnosis-contracts/docs/LMSRMarketMaker)
- Market (https://gnosis.github.io/gnosis-contracts/docs/Market)
- StandardMarket (https://gnosis.github.io/gnosis-contracts/docs/StandardMarket)

- StandardMarketFactory (https://gnosis.github.io/gnosis-contracts/docs/StandardMarketFactory)
- OlympiaToken (https://github.com/gnosis/olympia-token)

These are configured to use the web3 provider specified in Gnosis.create or subsequently modified with Gnosis.setWeb3Provider. The default gas costs for these abstractions are set to the maximum cost of their respective entries found in the *gas-stats.json* file built from the core contracts (https://github.com/gnosis/gnosis-contracts#readme). Additionally, the default message sender (i.e. *from* address) is set via the optional *defaultAccount* param in Gnosis.setWeb3Provider.

Gnosis.**create**(*opts*)

Factory function for asynchronously creating an instance of the API

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.ethereum** (*string|Provider*) – An instance of a Web3 provider or a URL of a Web3 HTTP provider. If not specified, Web3 provider will be either the browser-injected Web3 (Mist/MetaMask) or an HTTP provider looking at http://localhost:8545
- **opts.defaultAccount** (*string*) – The account to use as the default *from* address for ethereum transactions conducted through the Web3 instance. If unspecified, will be the first account found on Web3. See Gnosis.setWeb3Provider *defaultAccount* parameter for more info.
- **opts.ipfs** (*Object*) – ipfs-mini configuration object
- **opts.ipfs.host** (*string*) – IPFS node address
- **opts.ipfs.port** (*Number*) – IPFS protocol port
- **opts.ipfs.protocol** (*string*) – IPFS protocol name
- **opts.logger** (*function*) – A callback for logging. Can also provide 'console' to use *console.log*.

Returns **Gnosis** – An instance of the gnosis.js API

Gnosis.**defaultAccount**

The default account to be used as the *from* address for transactions done with this Gnosis instance. If there is no account, this will not be set.

Gnosis.**etherToken**

If on mainnet, this will be an EtherToken contract abstraction pointing to the MakerDAO WETH contract (https://etherscan.io/address/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2#code).

Otherwise, if EtherToken (https://gnosis.github.io/gnosis-contracts/docs/EtherToken/) is deployed to the current network, this will be set to an EtherToken contract abstraction pointing at the deployment address.

Gnosis.**lmsrMarketMaker**

If LMSRMarketMaker (https://gnosis.github.io/gnosis-contracts/docs/LMSRMarketMaker/) is deployed to the current network, this will be set to an LMSRMarketMaker contract abstraction pointing at the deployment address.

Gnosis.**olympiaAddressRegistry**

If AddressRegistry (https://github.com/gnosis/olympia-token) is deployed to the current network (this should only work for Rinkeby), this will be set to an AddressRegistry contract abstraction pointing at the deployment address. This is intended for use with Olympia.

Gnosis.**olympiaToken**

If OlympiaToken (https://github.com/gnosis/olympia-token) is deployed to the current network (this

should only work for Rinkeby), this will be set to an OlympiaToken contract abstraction pointing at the deployment address.

`Gnosis.`**`setWeb3Provider`**(*provider*, *defaultAccount*)
Setter for the ethereum web3 provider.

Note: this method is asynchronous and will return a Promise

Arguments

- **`provider`** (*`string|Provider`*) – An instance of a Web3 provider or a URL of a Web3 HTTP provider. If not specified, Web3 provider will be either the browser-injected Web3 (Mist/MetaMask) or an HTTP provider looking at http://localhost:8545
- **`defaultAccount`** (*`string`*) – An address to be used as the default *from* account for conducting transactions using the associated Web3 instance. If not specified, will be inferred from Web3 using the first account obtained by *web3.eth.getAccounts*. If no such account exists, default account will not be set.

`Gnosis.`**`standardMarketFactory`**
If StandardMarketFactory (https://gnosis.github.io/gnosis-contracts/docs/StandardMarketFactory/) is deployed to the current network, this will be set to an StandardMarketFactory contract abstraction pointing at the deployment address.

2.2 Methods

`calcLMSRCost`(*opts*)
Estimates the cost of buying specified number of outcome tokens from LMSR market.

Arguments

- **`opts.netOutcomeTokensSold`** (*`Array.<number>|Array.<string>|Array.<BigNumber>`*) – Amounts of net outcome tokens that have been sold. Negative amount means more have been bought than sold.
- **`opts.funding`** (*`number|string|BigNumber`*) – The amount of funding market has
- **`opts.outcomeTokenIndex`** (*`number|string|BigNumber`*) – The index of the outcome
- **`opts.outcomeTokenCount`** (*`number|string|BigNumber`*) – The number of outcome tokens to buy
- **`opts.feeFactor`** (*`number|string|BigNumber`*) – The fee factor. Specifying 1,000,000 corresponds to 100%, 50,000 corresponds to 5%, etc.

Returns **Decimal** – The cost of the outcome tokens in event collateral tokens

`calcLMSRMarginalPrice`(*opts*)
Estimates the marginal price of outcome token.

Arguments

- **`opts.netOutcomeTokensSold`** (*`Array.<Number>|Array.<string>|Array.<BigNumber>`*) – Amounts of net outcome tokens that have been sold. Negative amount means more have been bought than sold.
- **`opts.funding`** (*`number|string|BigNumber`*) – The amount of funding market has
- **`opts.outcomeTokenIndex`** (*`number|string|BigNumber`*) – The index of the outcome

Returns **Decimal** – The marginal price of outcome tokens. Will differ from actual price, which varies with quantity being moved.

calcLMSROutcomeTokenCount(*opts*)

Estimates the number of outcome tokens which can be purchased by specified amount of collateral.

Arguments

- **opts.netOutcomeTokensSold** (*Array.<Number>|Array.<string>|Array.<BigNumber>*) – Amounts of net outcome tokens that have been sold. Negative amount means more have been bought than sold.
- **opts.funding** (*number|string|BigNumber*) – The amount of funding market has
- **opts.outcomeTokenIndex** (*number|string|BigNumber*) – The index of the outcome
- **opts.cost** (*number|string|BigNumber*) – The amount of collateral for buying tokens

Returns **Decimal** – The number of outcome tokens that can be bought

calcLMSRProfit(*opts*)

Estimates profit from selling specified number of outcome tokens to LMSR market.

Arguments

- **opts.netOutcomeTokensSold** (*Array.<number>|Array.<string>|Array.<BigNumber>*) – Amounts of net outcome tokens that have been sold by the market already. Negative amount means more have been sold to the market than sold by the market.
- **opts.funding** (*number|string|BigNumber*) – The amount of funding market has
- **opts.outcomeTokenIndex** (*number|string|BigNumber*) – The index of the outcome
- **opts.outcomeTokenCount** (*number|string|BigNumber*) – The number of outcome tokens to sell
- **opts.feeFactor** (*number|string|BigNumber*) – The fee factor. Specifying 1,000,000 corresponds to 100%, 50,000 corresponds to 5%, etc.

Returns **Decimal** – The profit from selling outcome tokens in event collateral tokens

create(*opts*)

Factory function for asynchronously creating an instance of the API

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.ethereum** (*string|Provider*) – An instance of a Web3 provider or a URL of a Web3 HTTP provider. If not specified, Web3 provider will be either the browser-injected Web3 (Mist/MetaMask) or an HTTP provider looking at http://localhost:8545
- **opts.defaultAccount** (*string*) – The account to use as the default *from* address for ethereum transactions conducted through the Web3 instance. If unspecified, will be the first account found on Web3. See Gnosis.setWeb3Provider *defaultAccount* parameter for more info.
- **opts.ipfs** (*Object*) – ipfs-mini configuration object
- **opts.ipfs.host** (*string*) – IPFS node address

- **opts.ipfs.port** (*Number*) – IPFS protocol port
- **opts.ipfs.protocol** (*string*) – IPFS protocol name
- **opts.logger** (*function*) – A callback for logging. Can also provide 'console' to use *console.log*.

Returns **Gnosis** – An instance of the gnosis.js API

requireEventFromTXResult (*result*, *eventName*)

Looks for a single event in the logs of a transaction result. If no such events or multiple matching events are found, throws an error. Otherwise returns the matching event log.

Arguments

- **result** (*Transaction*) – Result of sending a transaction
- **eventName** (*string*) – Name of the event

Returns **Object** – The matching event log found

buyOutcomeTokens (*opts*)

Buys outcome tokens. If you have ether and plan on transacting with a market on an event which uses EtherToken as collateral, be sure to convert the ether into EtherToken by sending ether to the deposit() method of the contract. For other ERC20 collateral tokens, follow the token's acquisition process defined by the token's contract.

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.market** (*Contract|string*) – The market to buy tokens from
- **opts.outcomeTokenIndex** (*number|string|BigNumber*) – The index of the outcome
- **opts.outcomeTokenCount** (*number|string|BigNumber*) – Number of outcome tokens to buy
- **opts.limitMargin** (*number|string|BigNumber*) – Because transactions change prices, there is a chance that the cost limit for the buy, which is set to the cost according to the latest mined block, will prevent the buy transaction from succeeding. This parameter can be used to increase the cost limit by a fixed proportion. For example, specifying *limitMargin: 0.05* will make the cost limit increase by 5%.
- **opts.cost** (*number|string|BigNumber*) – Overrides the cost limit supplied to the market contract which is derived from the latest block state of the market along with the *outcomeTokenCount* and *limitMargin* parameters.
- **opts.approvalAmount** (*number|string|BigNumber*) – Amount of collateral to allow market to spend. If unsupplied or null, allowance will be reset to the *approvalResetAmount* only if necessary. If set to 0, the approval transaction will be skipped.
- **opts.approvalResetAmount** (*number|string|BigNumber*) – Set to this amount when resetting market collateral allowance. If unsupplied or null, will be the cost of this transaction.
- **opts.approveTxOpts** (*Object*) – Extra transaction options for the approval transaction if it occurs. These options override the options specified in sibling properties of the parameter object.
- **opts.buyTxOpts** (*Object*) – Extra transaction options for the buy transaction. These options override the options specified in sibling properties of the parameter object.

Returns **BigNumber** – How much collateral tokens caller paid

createCategoricalEvent(*opts*)
Creates a categorical event.

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.collateralToken** (*Contract | string*) – The collateral token contract or its address
- **opts.oracle** (*Contract | string*) – The oracle responsible for resolving this event
- **opts.outcomeCount** (*number | string | BigNumber*) – The number of outcomes of this event

Returns **Contract** – The created categorical event

createCentralizedOracle(*ipfsHash*)
Creates a centralized oracle linked to a published event.

Note: this method is asynchronous and will return a Promise

Arguments

- **ipfsHash** (*string*) – The published event's IPFS hash

Returns **Contract** – The created centralized oracle contract instance

createMarket(*opts*)
Creates a market.

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.event** (*Contract | string*) – The forwarded oracle contract or its address
- **opts.marketMaker** (*Contract | string*) – The collateral token contract or its address
- **opts.fee** (*number | string | BigNumber*) – The fee factor. Specifying 1,000,000 corresponds to 100%, 50,000 corresponds to 5%, etc.
- **opts.marketFactory** (*Contract | string*) – The factory contract

Returns **Contract** – The created market contract instance. If marketFactory is StandardMarketFactory (https://gnosis.github.io/gnosis-contracts/docs/StandardMarketFactory/), this should be a StandardMarket (https://gnosis.github.io/gnosis-contracts/docs/StandardMarket/)

createScalarEvent(*opts*)
Creates a scalar event.

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.collateralToken** (*Contract | string*) – The collateral token contract or its address
- **opts.oracle** (*Contract | string*) – The oracle responsible for resolving this event
- **opts.lowerBound** (*number | string | BigNumber*) – The lower bound for the event outcome

- **opts.upperBound** (*number|string|BigNumber*) – The upper bound for the event outcome

Returns **Contract** – The created scalar event

createUltimateOracle (*opts*)
Creates an ultimate oracle.

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.forwardedOracle** (*Contract|string*) – The forwarded oracle contract or its address
- **opts.collateralToken** (*Contract|string*) – The collateral token contract or its address
- **opts.spreadMultiplier** (*number|string|BigNumber*) – The spread multiplier
- **opts.challengePeriod** (*number|string|BigNumber*) – The challenge period in seconds
- **opts.challengeAmount** (*number|string|BigNumber*) – The amount of collateral tokens put at stake in the challenge
- **opts.frontRunnerPeriod** (*number|string|BigNumber*) – The front runner period in seconds

Returns **Contract** – The created ultimate oracle contract instance

loadEventDescription (*ipfsHash*)
Loads an event description from IPFS.

Note: this method is asynchronous and will return a Promise

Arguments

- **ipfsHash** (*string*) – The IPFS hash locating the published event

Returns **Object** – A POD object describing the event

publishEventDescription (*eventDescription*)
Publishes an event description onto IPFS.

Note: this method is asynchronous and will return a Promise

Arguments

- **eventDescription** (*Object*) – A POD object describing the event
- **eventDescription.title** (*string*) – A string describing the title of the event
- **eventDescription.description** (*string*) – A string describing the purpose of the event
- **eventDescription.resolutionDate** (*string*) – A string containing the resolution date of the event
- **eventDescription.outcomes** (*Array.<string>*) – A string array containing the outcomes of the event

Returns **string** – The IPFS hash locating the published event

resolveEvent (*opts*)
Resolves an event. Assumes event is backed solely by a centralized oracle controlled by you

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.event** (*Contract|string*) – The event address or instance
- **opts.outcome** (*number|string|BigNumber*) – The outcome to set this event to. This is the zero-based index of the outcome for categorical events and the decimals-adjusted value of the outcome for scalar events.

sellOutcomeTokens (*opts*)
Sells outcome tokens. If transacting with a market which deals with EtherToken as collateral, will need additional step of sending a withdraw(uint amount) transaction to the EtherToken contract if raw ether is desired.

Note: this method is asynchronous and will return a Promise

Arguments

- **opts.market** (*Contract|string*) – The market to sell tokens to
- **opts.outcomeTokenIndex** (*number|string|BigNumber*) – The index of the outcome
- **opts.outcomeTokenCount** (*number|string|BigNumber*) – Number of outcome tokens to sell
- **opts.limitMargin** (*number|string|BigNumber*) – Because transactions change profits, there is a chance that the profit limit for the sell, which is set to the profit according to the latest mined block, will prevent the sell transaction from succeeding. This parameter can be used to decrease the profit limit by a fixed proportion. For example, specifying *limitMargin: 0.05* will make the profit limit decrease by 5%.
- **opts.minProfit** (*number|string|BigNumber*) – Overrides the minimum profit limit supplied to the market contract which is derived from the latest block state of the market along with the *outcomeTokenCount* and *limitMargin* parameters.
- **opts.approvalAmount** (*number|string|BigNumber*) – Amount of outcome tokens to allow market to handle. If unsupplied or null, allowance will be reset to the *approvalResetAmount* only if necessary. If set to 0, the approval transaction will be skipped.
- **opts.approvalResetAmount** (*number|string|BigNumber*) – Set to this amount when resetting market outcome token allowance. If unsupplied or null, will be the sale amount specified by *outcomeTokenCount*.
- **opts.approveTxOpts** (*Object*) – Extra transaction options for the approval transaction if it occurs. These options override the options specified in sibling properties of the parameter object.
- **opts.sellTxOpts** (*Object*) – Extra transaction options for the sell transaction. These options override the options specified in sibling properties of the parameter object.

Returns **BigNumber** – How much collateral tokens caller received from sale

setWeb3Provider (*provider*, *defaultAccount*)
Setter for the ethereum web3 provider.

Note: this method is asynchronous and will return a Promise

Arguments

- **provider** (*string|Provider*) – An instance of a Web3 provider or a URL of a Web3 HTTP provider. If not specified, Web3 provider will be either the browser-injected Web3 (Mist/MetaMask) or an HTTP provider looking at http://localhost:8545
- **defaultAccount** (*string*) – An address to be used as the default *from* account for conducting transactions using the associated Web3 instance. If not specified, will be inferred from Web3 using the first account obtained by *web3.eth.getAccounts*. If no such account exists, default account will not be set.

Index