OptiGas README

The Problem

Transactions on the Ethereum blockchain require fees known as “gas”. The total cost of each transaction can be computed as: Gas used * Gas price. Because the pricing is based on supply and demand, the recommended gas price a user should use is constantly fluctuating. Exchanges are constantly interacting with the Ethereum network, so considering real-time prices matters to them just as much as they do to regular users on the network.

The Solution

An application that tracks trends and provides statistics for the gas prices on the Ethereum network would be a useful tool for an exchange (especially nowadays when they are so expensive).

1. Ingest gas price data from an API (Etherscan or EthGasStation)
2. Store data in a database
3. Expose a REST server with endpoints that return useful information

The Framework

Node.js

For the client/server interactions that will be involved in this project, Node.js is a platform that supports all the tools we need.

Express.js

We will need a back-end web app framework to set up a server that can hold our database and support our REST API. Express is the most popular framework used in Node.js for this purpose.

REST API

REST was chosen over Websocket because we are focused on GET requests to our server, and we are making occasional manual calls to our API rather than a continuous high-load interaction with it.

NeDB

We need a database that can store data and is easily queryable. NeDB is a lightweight Javascript database in which its datastore is the equivalent of a collection in MongoDB. Thus, we can utilize powerful MongoDB queries while maintaining a simple, lightweight framework. This is a balance of scalability and complexity considering the purpose of this application. If more time could be spent, using MySQL or MongoDB would be ideal to scale this project.

The Roadmap

Ingesting Data

First, we need to source our gas feed from an oracle. For this project, Etherscan was chosen over EthGasStation because it has historically provided more accurate estimates of gas prices and has a much more robust API. We need to obtain an API key by creating an account with a rate-limit of 1 req / 5 sec. On our server, we can use node-fetch to call https://api.etherscan.io/api?module=gastracker&action=gasoracle&apikey=YOURAPIKEY every 5 seconds. We can then parse the resulting JSON response for the main variables we are interested in: fast gas price, average gas price, low gas price, and block number. Along with this, we can record the current UNIX time of each new data entry with Javascript's Date.now() function. This will be useful later when we query prices between certain times from our database.

Database

After packing the desired variables into a JSON object, we insert them into our database. The database can be set up with NeDB's Datastore. Once we initialize a datastore, we can use its .insert() function to append new entries to our database. At this point, every 5 seconds we are calling the Etherscan API for gas prices, recording the time & relevant variables, and storing this in our database.

Endpoints

Next, the GET /gas implementation can be done with .get() from Express. We can use a MongoDB-style query on our database to get the most recent entry for prices. If we fail to do so, we return an error. Otherwise, we take this entry and strip the attributes that don't need to be seen on the user side (e.g. UNIX timestamp & ID in database). After this, we can return a JSON object containing the status of the action and the resulting data.

The result: `{ "error": false, "message": <JSON-object with fast, avg, and low gas prices and blockNum> }`

Likewise, we also implement the GET /average endpoint with .get() from Express. This endpoint will return the average gas price between two UNIX times. We make another MongoDB-style query using $gte and $lte with the bounds of our desired time interval. If we obtain an error, an empty query, or invalid parameters (e.g. not numbers, a future time), return the appropriate error message. Otherwise, the result is a collection of entries that were recorded during the specified time interval. We iterate through this collection and compute the average prices of the entries. Like before, bundle it into a new JSON object with its status.

The result: `{ "error": false, "message": <JSON-object with average gasPrice and time interval> }`

Docker

Finally, use Docker to containerize the project. Take special care when dealing with our dependency on node-fetch. To make things simple, append "type": "module" in the project's package.json file. In the Dockerfile, ensure npm install node-fetch is run after npm install -g npm and npm update.

Note: More specific details and explanations can be found in the code of server.js in the form of comments.

Setup and Use

Loading

After obtaining the docker image alanhwu/optigas, we can run it with the following command:

```
docker run -dp 9000:3000 alanhwu/optigas
```

The -p flag will run the server on the machine’s port 9000. Any other unused port can be used. Simply replace 9000 with the desired port. The -d flag will run the container in detached mode. This will be helpful so that we can run curl commands without opening a new terminal. For debugging purposes, we can see the console logs that are written when we ingest data or receive and confirm /average queries by running it with the interactive terminal using:

```
docker run -it -p 9000:3000 optigas-docker
```

Alternatively, after running it in detached mode, we can simply open the container in the Docker app to see the console.

Testing

Once we have this running, use commands in the following formats to test the endpoints:

- curl “localhost:<application_port>/gas”

  Example: curl “localhost:9000/gas”

  A response from the server indicating a failure will take the form of:

  `{ "error": true, "message": “Failed to fetch gas prices from database.” }`

  A successful response from the server will take the form of:

  `{ "error": false, "message":  <JSON-object with fast, avg, and low gas prices and blockNum>}`

  Example: `{"error":false,"message":{"fast":"84","average":"78","low":"77","blockNum":"13561368"}}`

Note: Provided gas prices are denoted in Gwei.

- curl “localhost:<application_port>/average?fromTime=<BeginningTime>&toTime=<EndTime>”

  Example: curl “localhost:9000/average?fromTime=1636182668&toTime=1636182688”

  A response from the server indicating a failure from an error will take the form of:

  `{ "error": true, "message": “Unable to calculate average. Ensure interval exists in database” }`

  A response from the server indicating a failure from **using a future UNIX time argument** will take the form of:

  `{"error":true,"message":"One or more arguments refers to a time in the future. Ensure interval exists in database"}`

  A successful response from the server will take the form of:

  `{ "error": false, "message": <JSON-object with average gas price, BeginningTime, EndTime>}`

  Example:
  `{"error":false,"message":{"averageGasPrice":73,"fromTime":1636182668,"toTime":1636182688}}`

Additional Notes: The API key remains as plain-text in the server.js file. This could be a potential security vulnerability. In Javascript, we can take multiple approaches to obfuscate the key in the files. However, regardless of how many levels of indirection we use or however else it is concealed locally, an API-request will eventually be made while running the application, and a malicious user can easily sniff out the key via a tool like Firebug. To fix this, we can have each user of the program use their own API keys and have a place for them to place it in for use.

Cheers!
- Alan