Building a GraphQL API for OpenSea Smart Contract Indexing

Jonathan Benghiat
Columbia University
M.S. in Data Science
New York, NY
jb4653@columbia.edu

Sienna Brent
Columbia University
B.A. in Electrical Engineering
New York, NY
scb2197@columbia.edu

Aman Chopra
Columbia University
M.S. in Data Science
New York, NY
ac5140@columbia.edu

Matthew Wang
Columbia University
B.A. in Computer Science
New York, NY
mw3071@columbia.edu

Anicia Yu
Columbia University
M.S. in Electrical Engineering
New York, NY
azy2107@columbia.edu

I. Introduction

Non-Fungible tokens, or NFTs, introduce a decentralized way of owning and controlling digital property. These tokens are digital assets with a unique identifier representing the ownership of various types assets such as collectibles, artwork, and music, and can be transferred when traded or sold. In 2017, they acquired a significant influence when the public started seeing them as a means of investment and began emerging more often on the Ethereum blockchain. Due to its fast growth and public attention, NFT marketplaces have become a popular place where creators can mint, sell, and auction NFTs.

The project aims to build a subgraph for OpenSea smart contract indexing using GraphQL API. This report will explain the purpose, methods, and uses of the subgraph that we created, which allows querying the OpenSea marketplace. Our code can be found here: https://github.com/jbenghiat830/NFT_data_warehouse

A. OpenSea

OpenSea is a decentralized marketplace for creating, buying, and selling NFTs on the Ethereum blockchain. Fully integrated with the most popular Ethereum wallets like MetaMask, the marketplace provides tools to help customers discover and purchase new and interesting NFTs. For creators and businesses, it allows people to customize storefronts, manage auctions, and set royalties for secondary sales, meaning that everytime the NFT is sold, the creator gets a small percentage of the sale. This results in OpenSea being the largest NFT marketplace anywhere, which has created a vibrant community between creators and collectors.

B. GraphQL

GraphQL is a query language for APIs that allows clients to specify what data they specifically need for APIs. It makes retrieving, sorting, and filtering data in a single request possible, and thus provides a powerful tool to building APIs which makes it very popular in many applications. Importantly, GraphQL provides the ability to create ”subgraphs”.

C. Subgraphs

A subgraph is a GraphQL endpoint that provides a way to access and filter data that is stored on the blockchain. Specifically in these applications, subgraphs are a way to define and query data from a specific or group of smart contracts on the blockchain and are a very important tool to developers building dApps. They function as APIs between the blockchain and the user, and importantly, developers can create their own subgraphs to define queries needed for their own dApps, allowing them to filter and tailor data to their own needs. The

behavior of the subgraph is specified in the subgraph manifest, and the subgraph is able to index the blockchain data in a way that follows the behavior. It can then be queried through a standard GraphQL API, with the specifics outlined in a schema file.

II. Purpose

A. Motivation

With such a large amount of data on OpenSea, creating a subgraph provides an efficient way to query and retrieve data from the Ethereum blockchain, which otherwise can be slow and expensive. These applications also become more scalable, flexible, and less complex. The Graph is a decentralized indexing protocol for querying the blockchain as well as networks like Ethereum and IPFS. As a result, a large part of the effort for this project was spent on creating our own subgraph which would then in turn be able to query the OpenSea marketplace for the event we specified, which is necessary to creating better user and developer experiences on the blockchain.

B. Use Cases

Subgraphs have many use cases, for example, a subgraph specifically programmed to index the OpenSea marketplace has the ability to track price history and trends of NFTs. It is useful, for example, to have a tool that can alert you when certain NFTs begin to price-climb at higher rates or conversely when an NFT you own or are considering owning begins to fall in value. Expanded beyond that, this tool would also be able to monitor the worth of an entire account on OpenSea including all the NFTs they currently own.

III. Implementation

A. Schema

The primary files we created in order to index our data efficiently and accurately were ”handler.ts”, ”schema.graphql” and ”subgraph.yaml”. The ”subgraph.yaml” file defines the details and basic features of our subgraph. Outlined in this file is the location for the schema file, as well as definitions for the data sources that the subgraph should draw from. In this section we defined the ABIs for Seaport, ERC165 type NFTs, and the NFTMetadata. These ABIs are needed in order to interact with the smart contract, and were directly extracted from EtherScan. Finally, we defined the event that we were looking for (OrderFulfilled) as well as the location to our handler for that event.

In the ”schema.graphql” file, we define the structure of the data that can be queried using GraphQL. It outlines the data model and types specific to OpenSea’s NFT marketplace that we intend to index, and also defines the available queries, mutations, and subscriptions that can be executed against the subgraph. Entities such as ”Collection” and ”Marketplace” are defined here, as well as many others which are viewable in the GitHub repository.

B. Procedure

The core feature of our OpenSea Indexing API was writing the handler for the ”OrderFulfilled” event, viewable in our GitHub repo in opensea-marketplace-indexer/src/handler.ts. We decided on focusing on this event in particular because it is emitted from the smart contract specifically when an NFT has been processed in a transaction. Thus, from this event we knew we would be able to extract lots of information for the index, such as NFT token ID, the valuation of an NFT, etc. The logic we used while writing this function is as follows:

The key to understanding an order event are the concepts of an ”offer” and a ”consideration”. The offer and considerations are simply lists of NFTs or money, on each side of the transaction. Our handler attempts to further define these two concepts by deducing which plays the role of the ”seller” (of the NFT) and which plays the role of the ”recipient” for any particular order.

To start, we process the ”Offer” and ”Consideration” fields of an incoming transaction, and determine which is the NFT seller and which is the NFT recipient. If the NFT is found to be contained within the ”offer”, then we know that the ”offerer” is the seller in the context of the transaction. On the flip side, if the ”consideration” contains the NFT, we then know that the ”recipient” is the seller of the transaction. This logic intuitively makes sense: if the offer parameter is found to not have an NFT in it then it must contain money, which is presumably being used to buy the NFT. It should be noted that our handler does not account for cases in which two (or more) NFTs are being traded rather than

bought. This logic was derived from the open-source Messari Seaport subgraph, available on GitHub [1].

After we have deduced the roles of the two users within a particular transaction, a "Sale" object is returned which has the corresponding assignments to buyer and seller as well as numeric information such as the NFT, the money and the fees.

In order to fill this "Sale" object with the correct information, however, a great deal of extraction needs to be done. First, let's assume we have a scenario such that the NFT is in the "consideration" and thus the recipient is the seller of the transaction. We start by extracting the NFT from the consideration, and the money amount from the offer. Performing the latter is relatively straightforward because in this scenario there is a single offerer with whom the entire money amount is directly associated. However, because there may be several consideration items received as part of the order along with the recipients of each item, we need to iterate through each one to retrieve the tokenIds and amounts of each NFT item. When one of these NFT items possesses a different token address from that of the first item, the entire item is thrown away – cases in which the consideration contains more than one token address happen, but are not handled by our code.

On the reverse of that scenario, where the NFT is in the "offer", both sets of items (those that are being offered and those being recieved) need to be iterated through and compounded to form a total sum associated with the overal sale. We extract the NFTs with an almost identical technique to before, except this time with offer items instead of consideration ones. Then, we use a very similar approach to extracting the money: we iterate through each item of the consideration, and if the consideration is considered "money" we add the money amount to a running total which is then returned to the higher-up function.

From the "Sale" object that is returned, we retrieve the token address of the NFT(s) and use it to obtain the corresponding collection. Using the attributes available in the NFT Metadata ABI, information about the NFT such as the name, symbol, and total supply can be retrieved from its token address alone. After the new collection (if it has not been seen before within the indexer's database) is instantiated, the marketplace whose ID we track in the Config is also instantiated and the collection count of the marketplace (in this case, OpenSea) is incremented by one.

Also using the information within the "Sale" object returned from the "getTransferDetails" function, we then create N new "Trades", one for each unique Token ID and amount present within the NFTs we extracted, either from the offer or the consideration. This "Trade" object is needed for each instance of an NFT because it stores extra information about the transaction, such as the method used to fulfill the order.

When the trades have been created, we update our collection and then our marketplace accordingly: The trade count of both the collection and marketplace we're tracking is then incremented by the same amount of new trades we instantiated, and the royalty fee of the collection and marketplace is updated as well. The account IDs of the buyer and seller we identified are stored, and the cumulative trade volume of Ethereum is updated accordingly.

Finally, once the statistics of both entities have been updated, we take a "snapshot" of the current state of each one. These "snapshots" are necessary because they associate all of the statistics we talked about and more with the real time at which the marketplace and the collection had that status. These are essential for being able to make sense of the incoming data on the user end, and the final task our handler does is to save both of these snapshots in the indexer's database.

C. Challenges

This project involved working with many new softwares, and as such our group faced a few challenges in implementing the code.

The main challenge we faced was learning AssemblyScript which was what the handler.ts was coded in. Because AssemblyScript is a strict variant of TypeScript, it was often difficult even coming from a TypeScript background to know which functionalities were able to be used, as well as understanding the new mechanisms which the new language brings.

Working with the Matchstick-As library [2] also proved to be difficult. MatchStick-As is a library used specifically for mocking features of The Graph

and subgraph entities, such as calls to and from smart contracts as well as the ”Events” which our handler was listening to. Because this library is relatively small and there is not much documentation, it was difficult to find help when a mocked function was not returning what it was supposed to.

IV. Results

A. Query Results

The images below are screenshots of an example query and its response.

```
query MyQuery {
  marketplaceDailySnapshots(orderBy: timestamp, orderDirection: desc, first: 10) {
    timestamp
    collectionCount
    tradeCount
    marketplaceRevenueETH
    creatorRevenueETH
    cumulativeUniqueTraders
  }
}
```

Fig. 1. Query 1


```
{
  "data": {
    "marketplaceDailySnapshots": [
      {
        "timestamp": "1667137343",
        "collectionCount": 23290,
        "tradeCount": 7869399,
        "marketplaceRevenueETH": "43970.908300865393798017",
        "creatorRevenueETH": "74791.923071181884786715",
        "cumulativeUniqueTraders": 954331
      },
      {
        "timestamp": "1667087999",
        "collectionCount": 23235,
        "tradeCount": 7847794,
        "marketplaceRevenueETH": "43887.507231276316803492",
        "creatorRevenueETH": "74619.896753909963152919",
        "cumulativeUniqueTraders": 951717
      },
      {
        "timestamp": "1667001599",
        "collectionCount": 23150,
        "tradeCount": 7806473,
        "marketplaceRevenueETH": "43738.176787282584821902",
        "creatorRevenueETH": "74387.893348037986406323",
        "cumulativeUniqueTraders": 947606
      },
```


Fig. 2. Response 1

Figure 1 and 2 showcase the daily summary of activity on OpenSea ordered by ”timestamp” (least to greatest) and is displayed in UNIX time. ”collectionCount” represents the number of NFTs, ”tradeCount” represents the number of transactions, ”marketplaceRevenueETH” represents the total commission which OpenSea receives, ”creatorRevenueETH” represents the royalties for the creator of a particular NFT, and finally, ”cumulativeUniqueTraders” represents the number of unique traders.

```
query MyQuery {
  collections(
    orderBy: totalRevenueETH
    orderDirection: desc
    first: 1
    where: {symbol: "BAYC"}
  ) {
    symbol
    cumulativeTradeVolumeETH
    nftStandard
    trades(first: 5, orderBy: timestamp, orderDirection: desc) {
      seller
      buyer
      timestamp
      transactionHash
      amount
      priceETH
    }
  }
}
```

Fig. 3. Query 2


```
{
  "data": {
    "collections": [
      {
        "symbol": "BAYC",
        "cumulativeTradeVolumeETH": "149522.225016887658608855",
        "nftStandard": "ERC721",
        "trades": [
          {
            "seller": "0x63a919d489db6608f9415553de2f2e417107cbf4",
            "buyer": "0xed2ab4948ba6a909a7751dec4f34f303eb8c7236",
            "timestamp": "1667184335",
            "transactionHash":
"0xd78b00c2db2a56702849c51e024d04357d509294202eb43748890e92e8932c01",
            "amount": "1",
            "priceETH": "62.62"
          },
          {
            "seller": "0xdf6a51fd9bb73a811d277e3181b4bfaacc1768a6",
            "buyer": "0xaa96a50a2f67111262fe24576bd85bb56ec65016",
            "timestamp": "1667123339",
            "transactionHash":
"0xd74dc68cc1e5cb2e7651dd17f18dab2f19097e660147e299c7e917a531b4fee2",
            "amount": "1",
            "priceETH": "75.5"
          },
          {
            "seller": "0xd7c38007bf425f57a4bc317c5d243843ebf72b9f",
            "buyer": "0xe2c12954a2dc21535a4d86b6d324bce51051c8ed",
            "timestamp": "1667112107",
            "transactionHash":
"0x96e3710ed0e075c84e24b47d5ff590f2b8eb3bee38d4e4f007c03d9d44896b69",
```


Fig. 4. Response 2

Figure 3 and 4 showcase the most recent transactions of Bored Ape Yacht Club token. The transaction hash, amount of tokens transferred, seller, buyer, and price are included the query response.

Figures 5 and 6 showcase the top 5 NFTs by trade volume. The names, symbols, royalty fee (percentage of money creator gets per transaction), and the cumulative amount of ETH traded for NFT are included in the query response.

```
query MyQuery {
  collections(
    orderBy: tradeCount
    orderDirection: desc
    first: 5
    where: {symbol_not: "null"}
  ) {
    name
    symbol
    royaltyFee
    cumulativeTradeVolumeETH
  }
}
```

Fig. 5. Query 3


```
{
  "data": {
    "collections": [
      {
        "name": "parallel",
        "symbol": "LL",
        "royaltyFee": "10",
        "cumulativeTradeVolumeETH": "4820.745954202173121552"
      },
      {
        "name": "Genuine Undead",
        "symbol": "GU",
        "royaltyFee": "2.5",
        "cumulativeTradeVolumeETH": "6872.338500109701636053"
      },
      {
        "name": "OpenSea Shared Storefront",
        "symbol": "OPENSTORE",
        "royaltyFee": "10",
        "cumulativeTradeVolumeETH": "35533.475899705164166614"
      },
      {
        "name": "Moonrunners",
        "symbol": "MOONR",
        "royaltyFee": "7.5",
        "cumulativeTradeVolumeETH": "8798.705639739638187372"
      },
      {
        "name": "y00ts Yacht Club",
        "symbol": "YYC",
        "royaltyFee": "5",
        "cumulativeTradeVolumeETH": "1653.96652541982350554"
      }
    ]
  }
}
```


Fig. 6. Response 3

It should be noted that our code is capable of other types of queries and is not limited to the one displayed.

B. Test Cases

In total, our group wrote 84 unit tests with full code coverage on 19 of the 20 total functions in our handler code. The function without full coverage is a parent function which only calls other functions, and as such we thought it would be redundant to write additional tests for it.

Fig. 7. Test case output

The structure of our test suite is divided first by the large umbrellas of the actual functions present within handler.ts, and then under each one the more granular features were each individually tested. For example, under the overall test for ”GetCollection()” [Fig. 7] we had two sub-tests, one asserting that when a collection does not exist, the function returns; the other asserting that when a collection does exist it loads an existing collection. The former of these tests was written by first checking that the collection we want doesn’t exist, then creating several mocked smart contract calls (with the matchstick-as framework) to control the return values and order

of execution. After this was finished the GetCollection() function was called, and the return values checked to ensure that it had in fact instantiated a new Collection. The latter of the two tests was created by first creating a collection at the address we want and filling in the attributes, then calling the testing function and comparing the returned values with the ones we had filled in.

Moving forward with expanding our library, we expect to expand our unit testing library as well, and with the same level of detail and caution. In writing our unit tests, we wanted each unit test to cover as small of a functionality as possible. By doing it this way, we knew that as the library becomes updated in the future it will become much quicker to pinpoint new bugs. Figures of the results from a few of our test cases are shown below.

V. Conclusion

The aim of this project was to create a tool capable of indexing the OpenSea NFT marketplace using the combination of a subgraph and the GraphQL API. Our product certainly achieves this outcome, as is proven by both the unit tests and the fully-functional demo of our subgraph which can be found hosted on The Graph.

This project was a long journey from start to finish, and may be described by a handful of milestones. The first of these was a comprehensive understanding of how The Graph works, as well as how to use it to index the Blockchain. We achieved this milestone by reading almost all of the documentation on the topic that we could find, as well as experimenting with subgraphs and reading their open-source code. Our second milestone was successfully getting the subgraph to index at all with the OpenSea marketplace, which we did by incorporating a large amount of external code while we were still too shaky on the concepts to write the entire thing ourselves. Once we achieved this, we were confident enough to customize the code we had into handling the specific event we wanted. Finally, our project was complete within the scope of this class when we finished writing the unit tests and achieved almost full coverage.

A. Contributions

Aman Chopra worked on the creation of the subgraph with Jonathan Benghiat, who wrote all 84 test cases. Sienna Brent, Matthew Wang, and Anicia Yu documented, compiled, and wrote this report.

References

[1] Messari, Messari. "Subgraphs/Mapping.ts at Master · Messari/Subgraphs." GitHub, 1 Nov. 2022, https://github.com/messari/subgraphs/blob/master/subgraphs/seaport/src.

[2] LimeChain. "Matchstick/Readme.Md at Main · Limechain/Matchstick." GitHub, 14 Feb. 2023, github.com/LimeChain/matchstick/blob/main/README.md.

[3] The Graph, thegraph.com/. Accessed 8 May 2023.

[4] Sharma, Rakesh. "Non-Fungible Token (NFT): What It Means and How It Works." Investopedia, 6 Apr. 2023, www.investopedia.com/non-fungible-tokens-nft-5115211.