**The Graph Protocol**

*If you are a developer in a team building DApp for end users, don’t miss this blog. You may want to use subgraphs in The Graph Network for your project. Let’s see why.*

1. **What is The Graph and why do we need The Graph?**
2. **How The Graph work**
3. **How to create and use subgraphs in a project**

**1) What is The Graph and why do we need The Graph?**

****

The Graph is an indexing protocol helping developers to access, combine and filter data on blockchain Ethereum, especially for complex data that cannot be accessed directly. It supports Ethereum, IPFS, POA Network,... and it keeps adding new blockchain.

Practically, as a developer, we can access data on blockchain easily via API Web3JS or EthersJS. However, it’s only useful with simple data such as the balance of an address or state variable in a smart contract. When you need to access complicated data on a blockchain, you will see that smart contracts are limited by their ability to effectively fetch data from the blockchain they are built on.



For example, the project Bored Ape Yacht Club is a collection of 10000 unique Bored Ape NFTs and any developer can get the information of each Ape easily via the getter function of its smart contract. But if he wants to get all Apes of a specific address or more advanced manipulation such as search, filter, check some relations, or get all Ape with specific attributes,.... the code will be much more complicated. Developers must spend extra time and money to manually gather that data themselves if they want to build a new DApp or improve the existing one. Many times, it may take several days to fetch all data from a smart contract because we need to fetch all events from all blocks to combine the data we need. Our app will have a lot of lag.

In other cases, applications such as blockchain explorer, for example, Etherscan built their service for reading all data on the blockchain and storing it in a database in a way that allows for quick retrieval of data. Developers can use similar services to get data for their DApp. But it is somehow ridiculous if a decentralized application uses data from centralized sources => your DApp can be manipulated by other people.

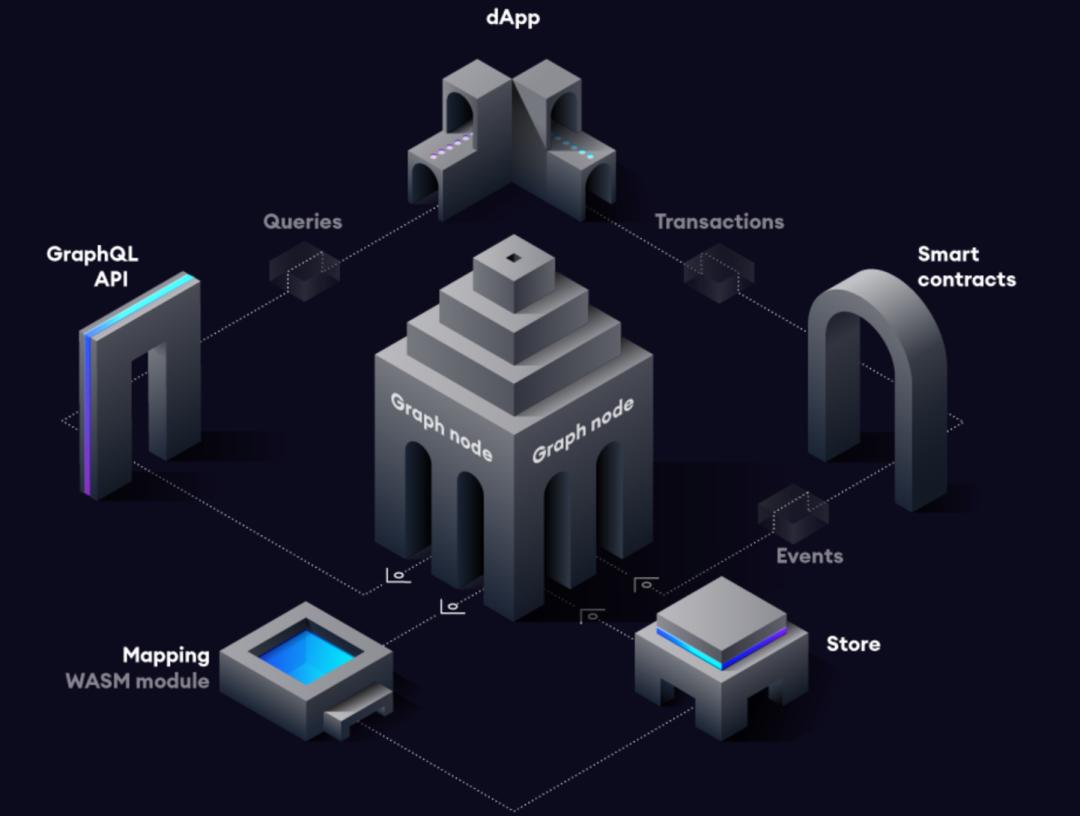
=> The Graph can solve this problem.

**2) How The Graph work**

A real-life example is an index in a book. Instead of going through the whole book page by page to find a concept we are looking for, we can find it much quicker in the index which is sorted alphabetically as it refers to the actual page in the book. It is like how Google indexes each website so we can easily search for information. That’s why The Graph is called an indexing protocol by helping developers get information sorted by each field easily on the blockchain.

The Graph supports hosted services in which we can create centralized API, and with the new version of The Graph, we can use decentralized APIs to access data with GRT token.

The Graph is like a marketplace for data where you can “‘buy” to access the API by spending token GRT. Each dataset is a subgraph like an open API defining which data you can take from the blockchain. You can access those ready decentralized APIs via the link: [Subgraphs | Graph Explorer (thegraph.com)](https://thegraph.com/explorer)

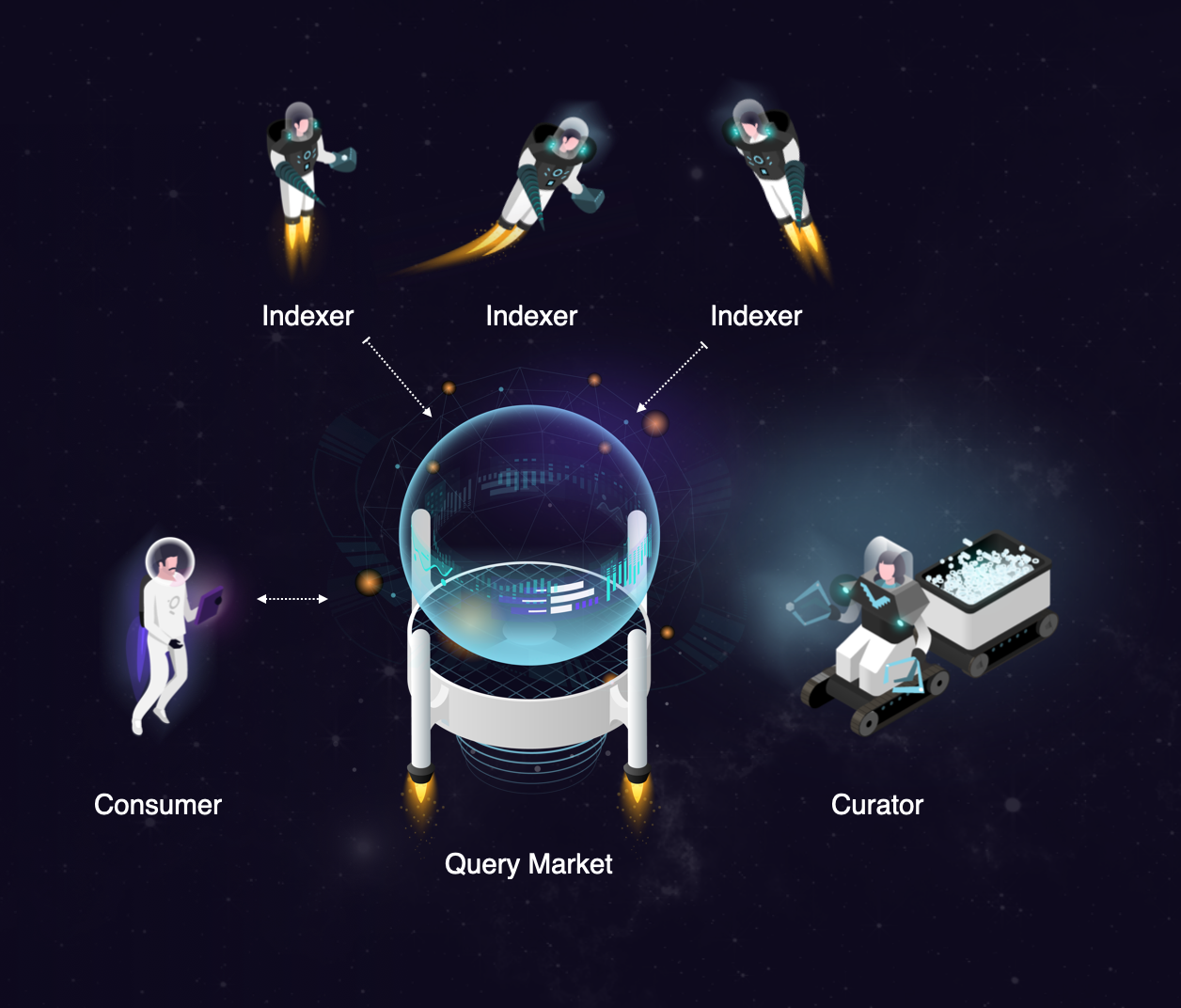
 To operate, The Graph has 4 roles: Indexer, Developer, Curator, and Delegator. For example, you want to make a DApp taking data from a blockchain, the procedure must follow the step below:

-> Developer make a subgraph define what you need from blockchain and then publish this subgraph to The Graph Network

-> Curators see that this subgraph is useful(many developers will need this subgraph to access data it defined), they will stake GRT token to the subgraph and signal that “this subgraph is worth indexing”.

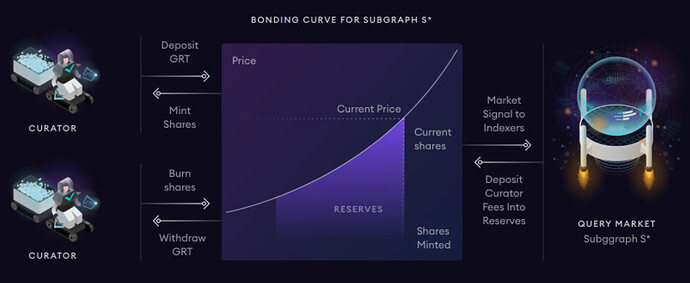
-> Indexer sees the signal and starts indexing that subgraph. Graph Node then finds all events in each block and runs handlers(WASM module) to update and then saves data to the store of Graph Node. This process may take a few days depending on the complexity of the data.

-> After that, customers can access data defined in the above subgraph using GraphQL query language and pay a querying fee. The data they get is fast and accurate in real-time.



There are ways to contribute to graph network for both technical and non-technical:

* Curator: after the developer makes a subgraph, curators will see whether this subgraph is worth indexing to signal to indexers. To make sure they always signal the highest quality subgraph, they must stake GRT token on a subgraph bonding curve.

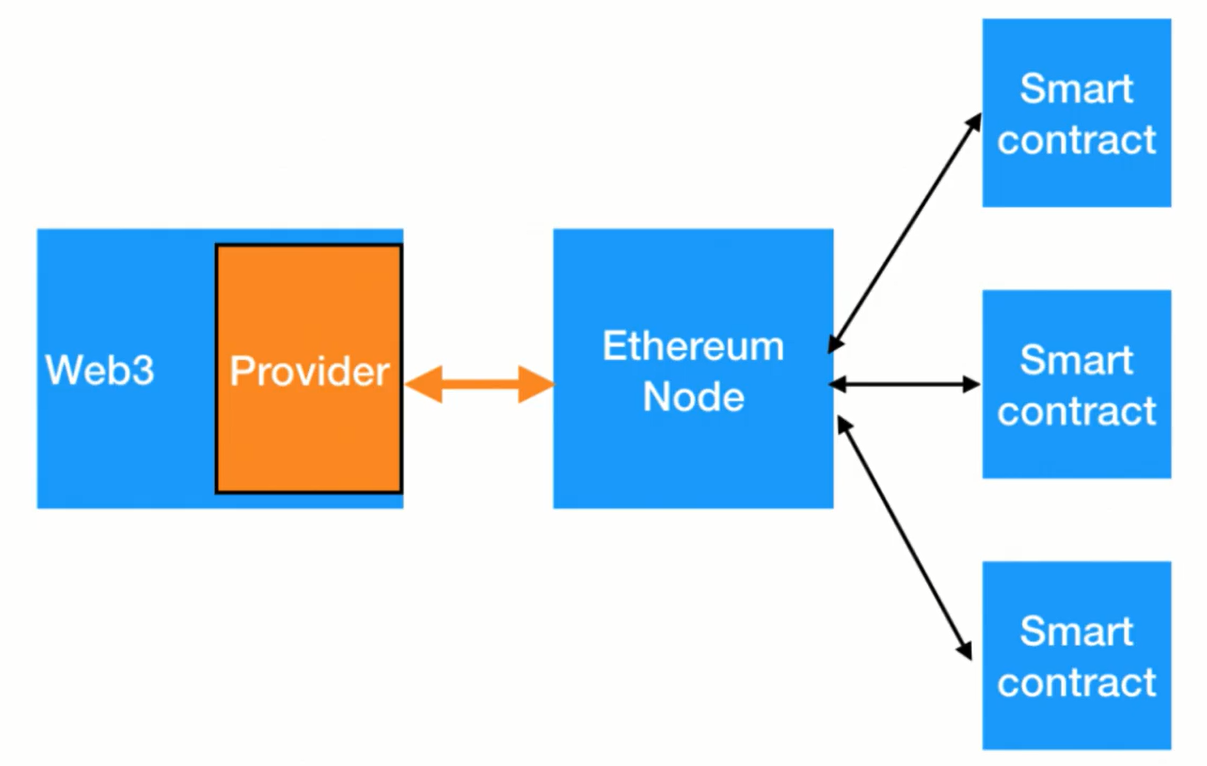


The bonding curve will give incentives to the curator to be the earliest curator staking on a new subgraph that they believe has high-quality information. This means that the sooner they signal, the more rewards they get in proportion to the amount of token they stake. If they move their stake off the subgraph, they must pay a withdrawal tax. Each time their subgraphs are queried by customers, they will earn part of the fee.

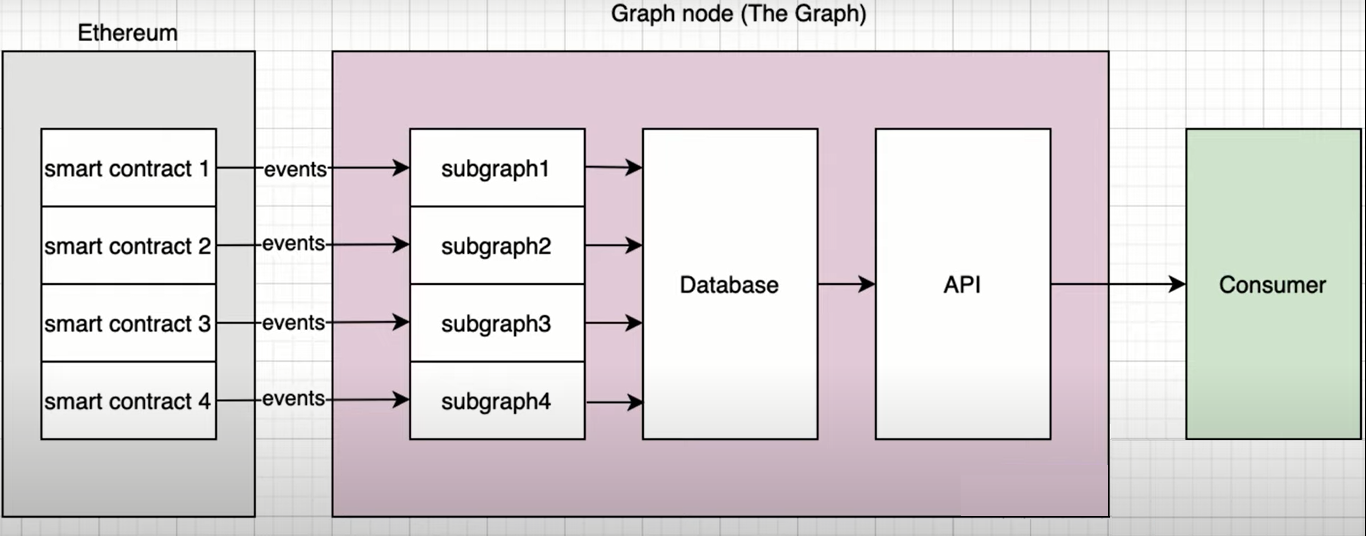
* Developer: they make subgraphs define which information to get from the blockchain. The Graph allows anyone to make subgraphs for The Graph Network. Developers are also usually curators because after making subgraphs, they often signal for their own subgraphs to the indexer.
* Indexer: they run Graph Node to index subgraphs from curators and handle queries from customers to send back the results. Each indexer runs Graph Node to form The Graph Network. To make sure they work correctly, they must stake GRT token to become an indexer. Therefore, if they provide incorrect data, part of their stake can be slashed. To operate a node, Indexers will have to invest capital and will also have to provide the necessary hardware and infrastructure to run Graph Node software.
* Delegator: the fee from the customer is decided by the indexer. To make sure indexers don’t overcharge customers, delegators will stake GRT to let the graph protocol know which indexer to use to fetch data for customers. They will choose the optimal indexer for the protocol. To become a delegator or curator, you must own enough GRT token.
* Customer: they use GraphQL in their project to query data from The Graph using their service. They must pay a querying fee, which is divided among delegators, indexers, curators. If the data they get from the indexer is incorrect, a dispute process between the customer and the indexer can be initiated.

In practice, if a project is highly potential and many people want to use data from it, the developer in the community is more likely to make a subgraph and start indexing to access its data because they can benefit from it. But if the project is not highly anticipated, the developer behind the project can become a curator to signal their own project to indexer and pay their own money. They can do that if they want to allow other developers access to their project’s data easily.

Compare architecture between using Web3 and The Graph:

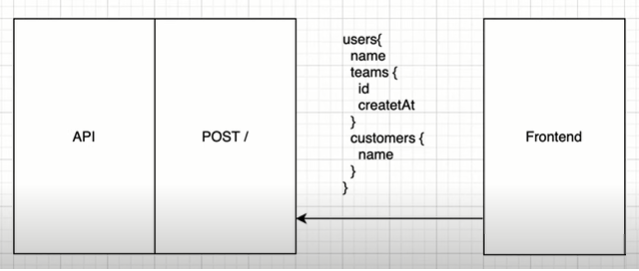


=> Using Web3, you can easily access simple data on the blockchain but if the data is complex, it is not good.



=> Using The Graph, the complicated things are done for you via subgraphs. Indexers have handled everything for you and saved them to the database in Graph Node. You just need to write some lines of code to fetch API from The Graph Network normally to get the data you want accurately in real-time.

For example, you can use The Graph API in your front end via GraphQL query:



Advantage:

* The Graph is an essential protocol when more and more projects are launched and the demand for interacting between them increases. It provides the optimal way for developers to get data from a blockchain.
* Anyone can join and create subgraphs to contribute to the protocol and gain profit.
* The Graph supports many blockchains and provides API for a lot of popular protocols such as Uniswap, Compound, AAVE,.. .and it keeps growing.

Disadvantage:

* Token GRT doesn’t have many utilities in the ecosystem outside of The Graph Network. There are still many protocols that The Graph hasn’t supported.

**3) How to create and use subgraph in a project**

*The link of all source code in this blog is at the end*

1. **Setup contract**

Here I have a simple smart contract:

| // SPDX-License-Identifier: UNLICENSED pragma solidity >=0.8.0; contract RandomAssignment{  mapping(address => uint) public valueOf;  event Assign(address indexed addr, uint indexed value);  function assignValueFor(address addr) public {  uint val = block.timestamp%15;  valueOf[addr] = val;  emit Assign(addr, val);  } } |
| --- |

In this contract we can set any address to a random value from 0 to 14, we can also change the value of the assigned address. An event is emitted each time the function assignValueFor is called. (Note: This is not the recommended way to generate a random value in solidity)

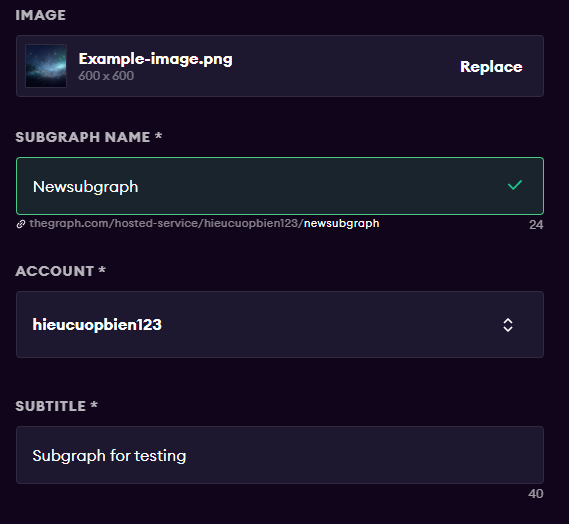
Now in my NodeJS project, we want to get a set of addresses that are assigned value more than 2 times in this smart contract. More complicated, we want to get a set of addresses assigned a value from 5 to 10 counting from the second address from block 10228285 forward sorted in ascending order of value. We also want to get data on how many addresses each value is assigned.

We can do it by Web3, but here we will use The Graph Protocol to query all this information much easier and faster in real-time.

For simplicity, we deploy this contract on the Rinkeby test net. You can use MetaMask as your wallet and have some ether on Rinkeby through the faucet: [Rinkeby Faucet](https://www.rinkebyfaucet.com/). You can use remix to deploy this contract for simplicity, please switch to injected Web3 to connect with your MetaMask in the Rinkeby test net. Here I deployed it at the address: 0x4a7EB54847FB807e9f1e39aaDea6e7f2daD61cfC and then I assigned some random values to some addresses using this contract (You can use this address for testing).

1. **Create centralized subgraph**

When we create a subgraph in hosted services of The Graph, we don’t need to own token GRT but create it for free. These are the supported networks on the Hosted Service: goerli, kovan, mainnet, matic, mumbai, poa-core, poa-sokol, rinkeby, ropsten, xdai. First, go to [Browse and Explore Subgraphs (thegraph.com)](https://thegraph.com/hosted-service/) -> Sign In -> My Dashboard -> Add Graph:



-> hit “Create subgraph”

If you don’t have NodeJS installed, you need to install it first.

-> Open folder project: “npm install -g @graphprotocol/graph-cli” to install graph-cli

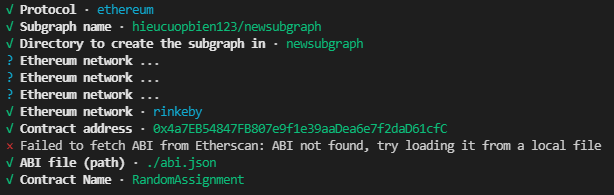
-> “graph init --contract-name <Contract name> --index-events --product <Product to create> --from-contract <address of contract>” to initiate project.

In this case, it must be: “graph init --contract-name RandomAssignment --index-events --product hosted-service --from-contract 0x4a7EB54847FB807e9f1e39aaDea6e7f2daD61cfC”

-> Then in the terminal, we choose subgraph’s name is <github’s username>/<name of subgraph in lower case>

In my case, it must be: hieucuopbien123/newsubgraph

We also set other information:



If you haven’t published your code, you must provide the ABI file of your contract. Here I take the ABI file when compiling the contract on Remix and paste it to file abi.json

This will generate a basic subgraph based on the contract address passed in as the argument to --from-contract. By using this contract address, the CLI will initialize a few things in your project to get you started (including fetching the ABIs and saving them in the ABIs directory).

By passing the --index-events flag, the CLI will automatically populate some code and configuration for us in the files schema.graphql, subgraph.yaml, and src/mapping.ts based on the events emitted from the contract. We just need to modify these files to get our data

-> “cd newsubgraph”

-> “npm install” to install library graph-ts for your project. You need this library to code your typescript file

-> In file schema.graphql, we define each field(schema) of data that we want to get:

| type AddressValue @entity {  id: ID!  value: BigInt!  count: Int! } type Address @entity {  id: ID!  timesChange: Int!  blockNumber: BigInt!  timestamp: BigInt!  currentValue: BigInt! } |
| --- |

We need to define 2 entities: Address and AddressValue. We want to get how many addresses each value is assigned, so it is the field “count” in “AddressValue”. In “Address”, we want to count how many times the address has been assigned a value(timeChanged), currentValue, blockNumber, and timestamp. The field “id” with the type “ID!” is required in every entity.

->In file subgraph.yaml, we write configuration for the subgraph like the entities, startBlock, event to catch:

| source:  address: "0x4a7EB54847FB807e9f1e39aaDea6e7f2daD61cfC"  abi: RandomAssignment  startBlock: 10229407 |
| --- |

Here we define the startBlock as the number of the block that the data source starts indexing from. In most cases, we suggest using the block in which the contract was created, if not then the indexer will start from the genesis block.

We also define which entities and events to index:

| entities:  - AddressValue  - Address |
| --- |

| eventHandlers:  - event: Assign(indexed address,indexed uint256)  handler: handleAssign |
| --- |

It means when the event “Assign” is emitted from the smart contract, it will update data by calling the function “handleAssign” that we define below

-> “graph codegen”

To make smart contracts, events, and entities easy and type-safe, the Graph CLI generates AssemblyScript types from a combination of the subgraph's GraphQL schema and the contract ABIs included in the data sources. This command creates a new folder generated containing AssemblyScript.

-> Code the mapping.ts file to determine how the data is obtained from the blockchain:

| import { Assign as AssignEvent } from "../generated/RandomAssignment/RandomAssignment" import { AddressValue, Address } from "../generated/schema"; |
| --- |

We need to import all entities defined in schema.graphql to get data for them in this file. The Graph will run from the startBlock to the newest block, catch all events chronologically and handle it via function in mapping.ts

| export function handleAssign(event: AssignEvent): void {  let val = AddressValue.load(event.params.value.toString());  if(!val){  val = new AddressValue(event.params.value.toString());  val.value = event.params.value;  val.count = 0;  }    let address = Address.load(event.params.addr.toHexString());  if(!address){  val.count++;  address = new Address(event.params.addr.toHexString());  address.timesChange = 0;  }  address.blockNumber = event.block.number;  address.timesChange++;  address.timestamp = event.block.timestamp;  address.currentValue = event.params.value;  address.save();  val.save(); } |
| --- |

When event Assign comes, we will get the entity AddressValue via function AddressValue.load(<id of entity>); and check if it doesn’t exist(it is the first time an address is assigned this value), we will initiate this entity. Similarly, with entity Address, we initiate and write code to update each entity. Finally, we save them via function save()

-> “graph auth --product hosted-service <ACCESS\_TOKEN>” with access token is in your subgraph in the website. In my case:

“graph auth --product hosted-service f7dbe86bf45d404eaada5123534fc771”



-> run “graph deploy --product hosted-service <github user>/<subgraph name>“ to deploy subgraph to Hosted Service

In my case, it must be: “graph deploy --product hosted-service hieucuopbien123/newsubgraph”

-> After deployment, we need to wait for The Graph to sync data on the blockchain, and then we can use it in our DApp.

Note: there is no curator or indexer here because it is a centralized API.

1. **Create decentralized subgraph**

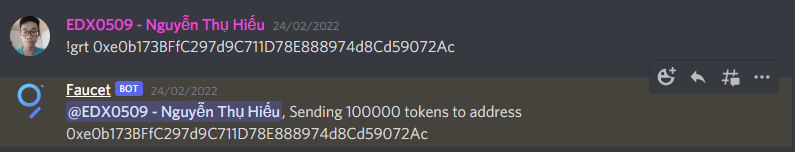
It is similar to creating a centralized subgraph. Go to [My Subgraphs | Subgraph Studio (thegraph.com)](https://thegraph.com/studio/) -> connect MetaMask to accept using The Graph -> Create a Subgraph -> we can create a subgraph following those above commands.

Note:

* We must use “npm install @graphprotocol/graph-ts” to install the graph-ts library manually if the file package.json in our subgraph doesn’t include it.
* Our subgraph’s name in the command must be our subgraph’s name that we created in Subgraph Studio in lowercase, which does not contain your Github's username like in hosted service because we don’t need to sign in to our Github account here. Everything we need is a decentralized wallet like Metamask.
* Instead of using access\_token, we use deploy\_key

After deployment and sync in your subgraph studio, we can publish our subgraph into The Graph Network to share our API with other people. And then we can become a curator by staking GRT token to our subgraph and signaling it to attract indexers.

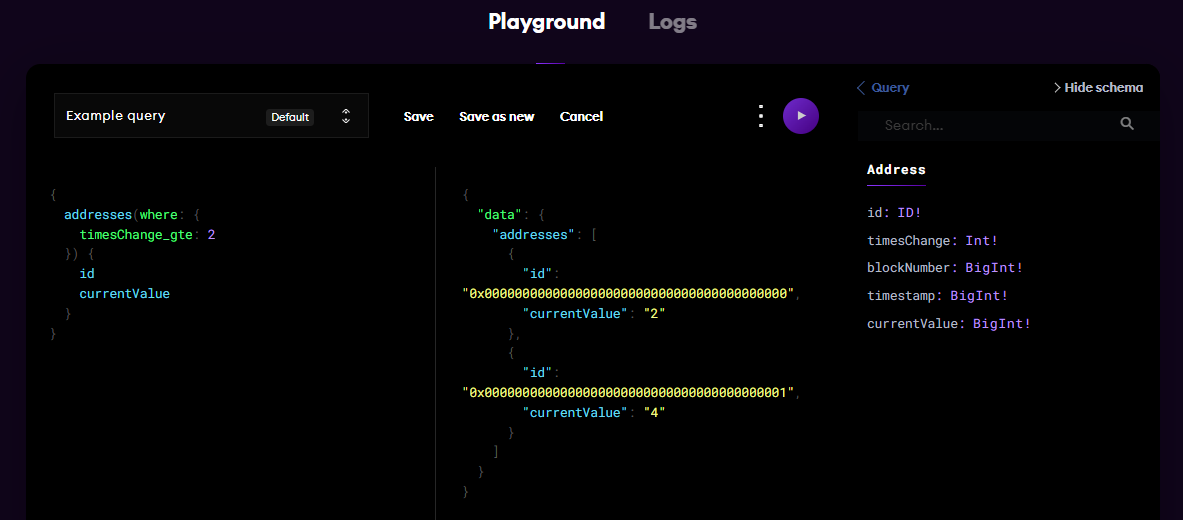
You can get token GRT to your wallet on the Rinkeby test net by joining discord of The Graph at link: <https://discord.com/invite/vtvv7FP> and get the role Testnet Indexer and then request GRT token to your address in channel 🚰-testnet-faucet by command:



After indexing, other people can access to your API

1. **Use API from subgraph in your project**

- We can query data via GraphQL in the Playground section of The Graph for testing purposes. For example, we query “a set of addresses that are assigned value more than 2 times” by GraphQL:



=> The first column is our query, the second column is output, the third column is some schema that we can use in our GraphQL queries.

- To use data in our NodeJS project, we need an URL. With subgraphs in hosted service, we can access the data any time via the ready HTTP URL. With subgraphs in The Graph Explorer, after indexing, we need to create an API key in Subgraph Studio to get access to the URL to query data that will cost you GRT token to query data.

Let’s use the API in hosted service that we created in section c) for free

-> Open our subgraph to get the url. In my case, the url is: <https://api.thegraph.com/subgraphs/name/hieucuopbien123/newsubgraph>

-> “npm init -y” to init a NodeJS project

-> “npm install axios”

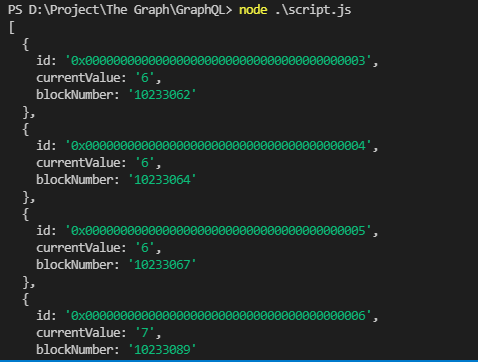
-> create file script.js

| const axios = require('axios'); const main = async () => {  try{  const result = await axios.post( "https://api.thegraph.com/subgraphs/name/hieucuopbien123/newsubgraph",  {  query:`{  addresses(orderDirection: asc,  orderBy: currentValue,  skip: 1  where:{ blockNumber\_gt: 10228285,  currentValue\_lte: 10,  currentValue\_gte: 5}) {  id  currentValue  blockNumber  }  }`  }  );  console.log(result.data.data.addresses);  }catch(error){  console.log(error);  } } main(); |
| --- |

Here I get “a set of addresses with current value from 5 to 10 counting from the second address from block 10228285 forward sorted in ascending order of value” easily with a single call API.

Note: when using GraphQL to query data in The Graph, we must use the POST API although we just read data.

-> “node script.js” to run this script to get data. In this case:



Link code: [hieucuopbien123/TestUsingSubgraph (github.com)](https://github.com/hieucuopbien123/TestUsingSubgraph)

Reference:

[Tại sao lại cần Subgraph? – The GraphVN](https://thegraphvn.com/tai-sao-lai-can-subgraph/)

[The Graph là gì? Toàn bộ thông tin về The Graph và token GRT | Coinvn](https://coinvn.com/the-graph-la-gi-toan-bo-thong-tin-ve-the-graph-va-token-grt/)

[Building Subgraphs with Subgraph Studio (thegraph.com)](https://thegraph.com/blog/building-with-subgraph-studio)

[Query Ethereum with GraphQL with The Graph - YouTube](https://www.youtube.com/watch?v=l2rzT_Dp4T0)