

Lesson 12 - Foundry / The Graph

Today's topics

- Foundry setup
- Foundry testing
- The Graph

There's a bit of a trend toward writing "clever" Solidity code and it's not a good one. Boring code is better code.

Quote

Foundry is a smart contract development toolchain written in Solidity.

Forge is the CLI tool to initialise, build and test Foundry projects.

Foundry allows fast (milliseconds) and sophisticated Solidity tests, e.g. reading and writing directly to storage slots.

Getting Started

There is an example project in our [repo](#)

1. Install Foundry
 - [Steps for various operation systems.](#)
2. Initialise a project called **lets_forge**
 - `forge init lets_forge`
3. Build the project
 - `forge build`
4. Verify install has worked by running the tests
 - `forge test`

Folders `out` and `cache` have been generated to store the contract artifact (ABI) and cached data, respectively.

Selecting solc version used by system: <https://github.com/crytic/solc-select>.

Testing: Basics

Smart contracts are tested using smart contracts, which is the secret to Foundry's speed since there is no additional compilation being carried out.

A smart contract e.g. `MyContract.sol` is tested using a file named `MyContract.t.sol`:

```
|—— src
|  |—— MyContract.sol
|  |—— test
|  |—— MyContract.t.sol
```

`MyContract.t.sol` will import the contract under test in order to access its functions.

First Contract

Create a contract called `A.sol` and save it in `src` with the following contents:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

/// @title Encode smart contract A
/// @author Extropy.io
contract A {
    uint256 number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256) {
        return number;
    }
}
```

Then create a file to test the smart contract, for example `A.t.sol` in `test` with the following contents:

```

// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

// Standard test libs
import "forge-std/Test.sol";
import "forge-std/Vm.sol";

// Contract under test
import {A} from "../src/A.sol";

contract ATest is Test {
    // Variable for contract instance
    A private a;

    function setUp() public {
        // Instantiate new contract instance
        a = new A();
    }

    function test_Log() public {
        // Various log examples
        emit log("here");
        emit log_address(address(this));
        // HEVM_ADDRESS is a special reserved address for the VM
        emit log_address(HEVM_ADDRESS);
    }

    function test_GetValue() public {
        assertTrue(a.retrieve() == 0);
    }

    function test_SetValue() public {
        uint256 x = 123;
        a.store(x);
        assertTrue(a.retrieve() == 123);
    }

    // Define the value(s) being fuzzed as an input argument
    function test_FuzzValue(uint256 _value) public {
        // Define the boundaries for the fuzzing, in this case 0 and 99999
        _value = bound(_value, 0, 99999);
        // Call contract function with value
        a.store(_value);
        // Perform validation
        assertTrue(a.retrieve() == _value);
    }
}

```

Finally run the test with `forge test -vv` to see the results and all logs.

Run tests

```
forge test
```

To print event logs:

```
forge test -vv
```

To print trace of any failed tests:

```
forge test -vvv
```

To print trace of all tests:

```
forge test -vvvv
```

To run a specific test:

```
forge test --match-test test_myTest
```

Dependencies

Forge uses [git submodules](#) so it works with any GitHub repo that contains smart contracts.

Adding Dependencies

To install e.g. OpenZeppelin contracts from the [repo](#) one would run

```
forge install OpenZeppelin/openzeppelin-contracts.
```

The repo will have been cloned into the `lib` folder.

Dependencies can be updated by running `forge update`.

Removing Dependencies

Dependencies can be removed by running

```
forge remove openzeppelin-contracts,
```

which is equivalent to

```
forge remove OpenZeppelin/openzeppelin-contracts.
```

Integrating with Existing Hardhat project

Foundry can work with Hardhat-style projects where dependencies are npm packages (stored in `node_modules`) and contracts are stored in `contracts` as opposed to `source`.

1. Copy lib/forge-std from a newly-created empty Foundry project to this Hardhat project directory.
2. Copy foundry.toml configuration to this Hardhat project directory and change src, out, test, cache_path in it:

```
[default]
src = 'contracts'
out = 'out'
libs = ['node_modules', 'lib']
test = 'test/foundry'
cache_path = 'forge-cache'
```

3. Create a remappings.txt to make Foundry project work well with VS Code Solidity extension:

```
ds-test/=lib/forge-std/lib/ds-test/src/
forge-std/=lib/forge-std/src/
```

4. Make a sub-directory test/foundry and write Foundry tests in it.

Foundry test works in this existing Hardhat project. As the Hardhat project is not touched and it can work as before.

See folder `hardhat-foundry`. Both test suites function:

```
npx hardhat test
```

```
forge test -vvv
```

Deploying Contracts

`forge create` allows you to deploy contracts to a blockchain.

To deploy the previous contract to e.g. Ganache using an account with private key `4e0...9b` you would type:

```
forge create A --legacy --contracts src/A.sol --private-key 4e0...9b --rpc-url
http://127.0.0.1:8545
```

Result:

```
Deployer: 0x536f8222c676b6566ff34e539022de6c1c22cc06
Deployed to: 0x79bb7a73d02b6d7e2e98848d26ad911720421df0
Transaction hash:
0xc5bb34ee82dc2f57bd7f7862eec440576a7cc7cfe4533392192704fd44653b68
```

The `--legacy` flag is used since Ganache doesn't support EIP-1559 transactions. Hardhat (`npm run hardhat node`) circumvents this issue.

Interacting with Contracts

```
cast call 0x79bb7a73d02b6d7e2e98848d26ad911720421df0 "retrieve()" --rpc-url
http://127.0.0.1:8545
```

Testing: Advanced

Sometimes it's useful to test a function with many input variables at random in order to test edge-cases. This is called `fuzz testing`, or simply `fuzzing`.

An example of fuzzing using Foundry has been provided in the smart contract:

```
// Define the value(s) being fuzzed as an input argument
function test_FuzzValue(uint256 _value) public {
    // Define the boundaries for the fuzzing, in this case 0 and 99999
    _value = bound(_value, 0, 99999);
    // Call contract function with value
    a.store(_value);
    // Perform validation
    assertTrue(a.retrieve() == _value);
}
```

Interpreting results

Results may appear similar to this:

```
Running 4 tests for test/A.t.sol:ATest
[PASS] test_FuzzValue(uint256) (runs: 256,  $\mu$ : 31708,  $\sim$ : 32330)
[PASS] test_GetValue() (gas: 7546)
[PASS] test_Log() (gas: 3930)
Logs:
here
0xb4c79dab8f259c7aee6e5b2aa729821864227e84
0x7109709ecfa91a80626ff3989d68f67f5b1dd12d
```

- "runs" refers to the amount of scenarios the fuzzer tested. By default, the fuzzer will generate 256 scenarios, however, this can be configured using the FOUNDRY_FUZZ_RUNS environment variable.
- " μ " (Greek letter mu) is the mean gas used across all fuzz runs.
- " \sim " (tilde) is the median gas used across all fuzz runs.

General information

Full suite of logging and assertions: <https://book.getfoundry.sh/reference/ds-test.html>

| *msg.sender is address(this) i.e. the testing contract*

- Is not payable - must deploy with another address e.g. if owner is payable

| *Change msg.sender*

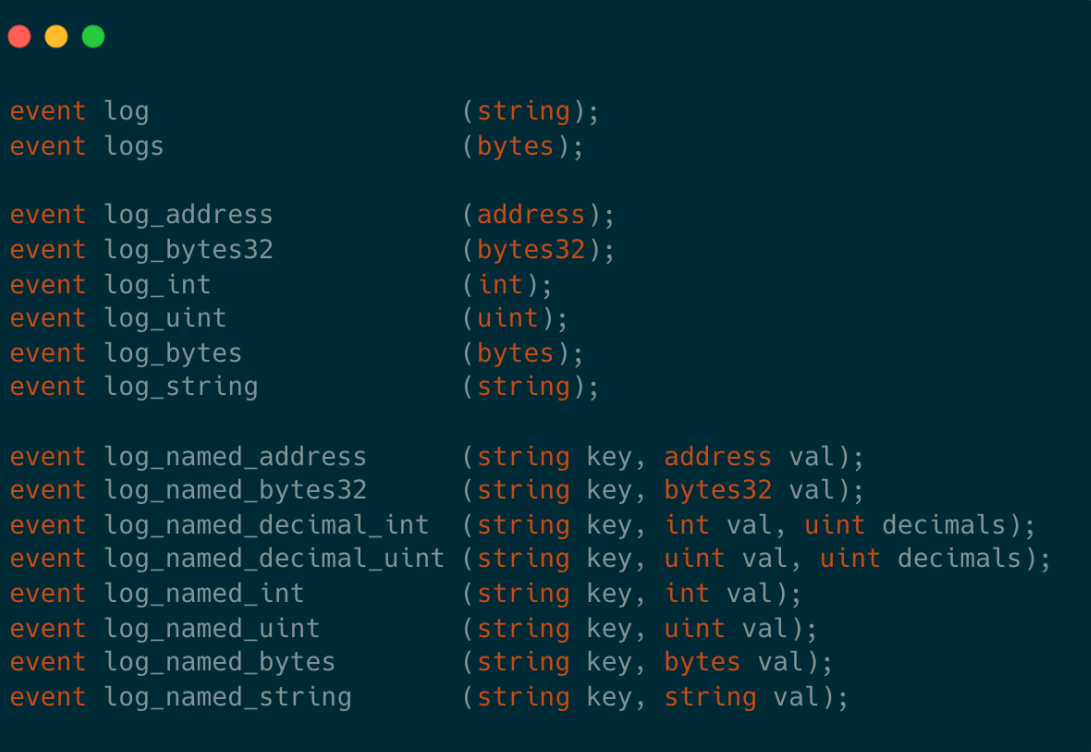
```
vm.prank(address(1));
```

```
vm.prank(address(0xf71840e248674E2Bb49427868810506d091aAD1e));
```

| *Label addresses in test trace*

```
vm.label(owner, "This is the owner");
```

Default available event logs:



```
event log                (string);
event logs               (bytes);

event log_address        (address);
event log_bytes32        (bytes32);
event log_int            (int);
event log_uint           (uint);
event log_bytes          (bytes);
event log_string         (string);

event log_named_address  (string key, address val);
event log_named_bytes32  (string key, bytes32 val);
event log_named_decimal_int (string key, int val, uint decimals);
event log_named_decimal_uint (string key, uint val, uint decimals);
event log_named_int      (string key, int val);
event log_named_uint     (string key, uint val);
event log_named_bytes    (string key, bytes val);
event log_named_string   (string key, string val);
```

Useful Foundry references:

<https://0xkwoon.substack.com/p/learn-enough-foundry-to-be-dangerous?s=r>

<https://chainstack.com/foundry-a-fast-solidity-contract-development-toolkit/>

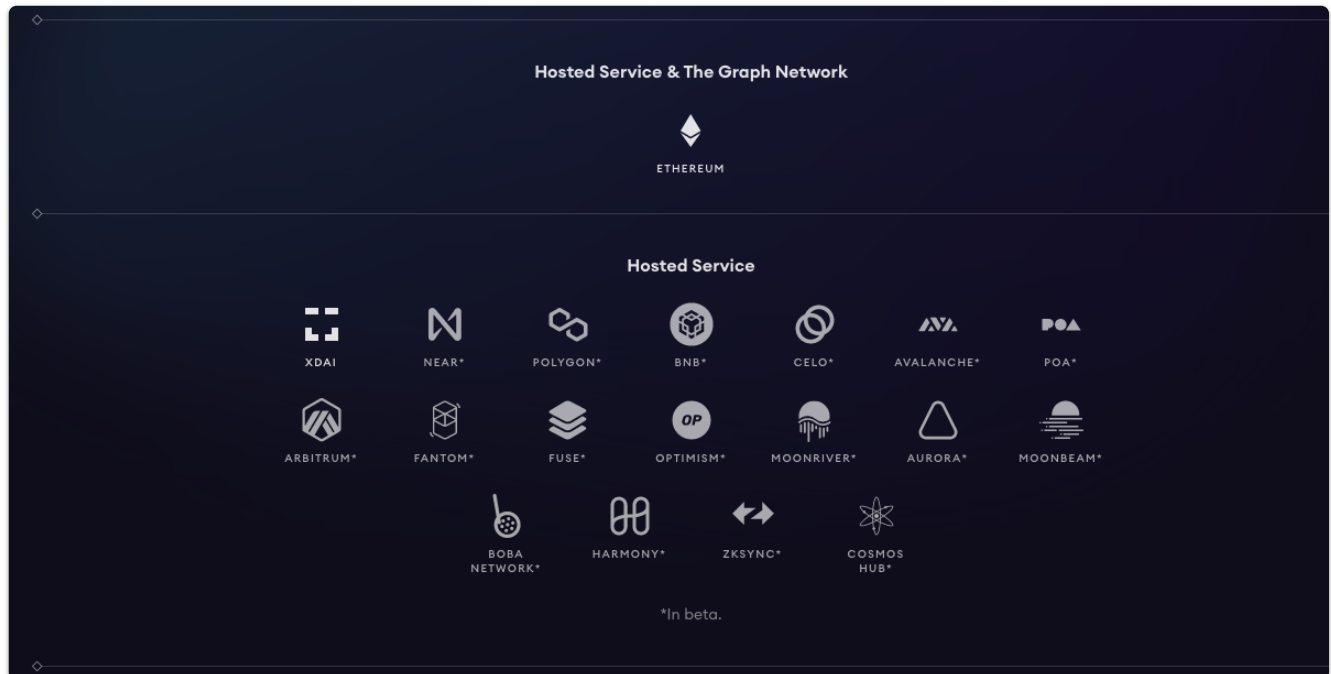
<https://book.getfoundry.sh/reference/ds-test?highlight=log#logging>

The Graph

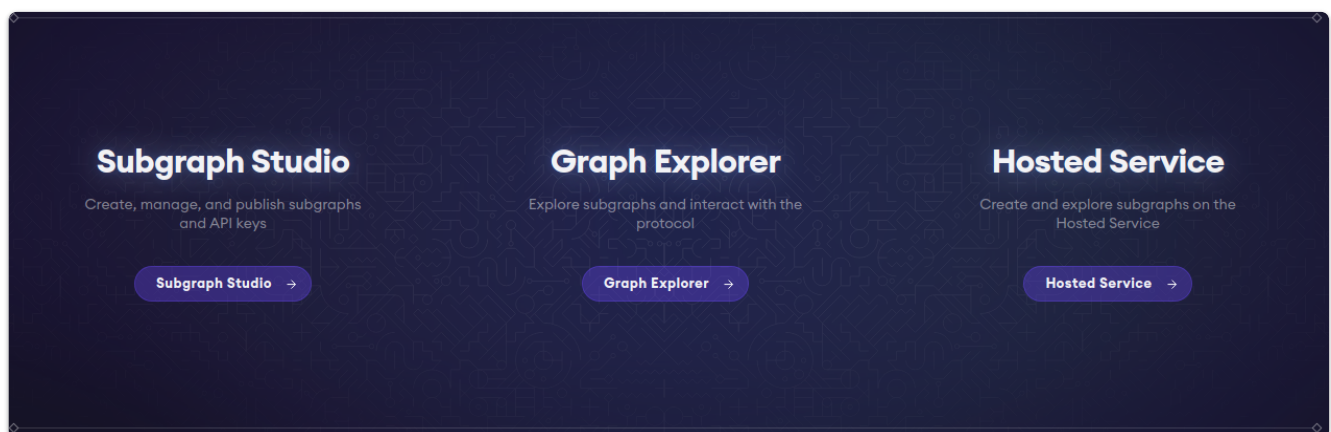
The Graph is an indexing protocol for querying networks like Ethereum and IPFS. Anyone can build and publish open APIs, called subgraphs, making data easily accessible.

See [The Graph Academy](#)


Supported Networks



Products



Hosted Service

COMPOUND-FINANCE

Compound V2

🔖 111

• Syncing (100%)

14.0M / 14.0M blocks

Compound is an open-source protocol for algorithmic, efficient Money Markets on the Ethereum blockchain.

Network	Last updated	Created	Entities
mainnet	2 months ago	2 years ago	6,372,155

Github
<https://github.com/graphprotocol/compound-V2-subgraph>

ID
QmfKmZ7xFT9PRydVGJRRWEdr7rbBcRoKbDyX7fGK7MCq5j

Queries (HTTP)
<https://api.thegraph.com/subgraphs/name/graphprotocol/compound-v2>

cDAI Hodlers

⏮ ⏪ ⏩ ⏭

```
{
  accountCTokens(where: {symbol: "cDAI"}) {
    id
    symbol
    cTokenBalance
    accountBorrowIndex
    totalUnderlyingSupplied
    totalUnderlyingRedeemed
    totalUnderlyingBorrowed
    totalUnderlyingRepaid
    lifetimeSupplyInterestAccrued
    lifetimeBorrowInterestAccrued
    borrowBalanceUnderlying
    supplyBalanceUnderlying
    storedBorrowBalance
  }
}
```

```
"0x5d3a536e4d6dbd6114cc1ead35777bab948e3643-
0x000000000008c4fb1c916e0c88fd4cc402d935e7d",
  "symbol": "cDAI",
  "cTokenBalance": "0.00000002",
  "accountBorrowIndex": "0",
  "totalUnderlyingSupplied":
    "638004.383717739635441252",
  "totalUnderlyingRedeemed":
    "638004.38371773920172639",
  "totalUnderlyingBorrowed": "0",
  "totalUnderlyingRepaid": "0",
  "lifetimeSupplyInterestAccrued":
    "2.55468072468302e-12",
  "lifetimeBorrowInterestAccrued": "0",
  "borrowBalanceUnderlying": "0",
  "supplyBalanceUnderlying":
    "4.3626954272468302e-10",
  "storedBorrowBalance": "0"
```

< Schema > Hide schema

Search...

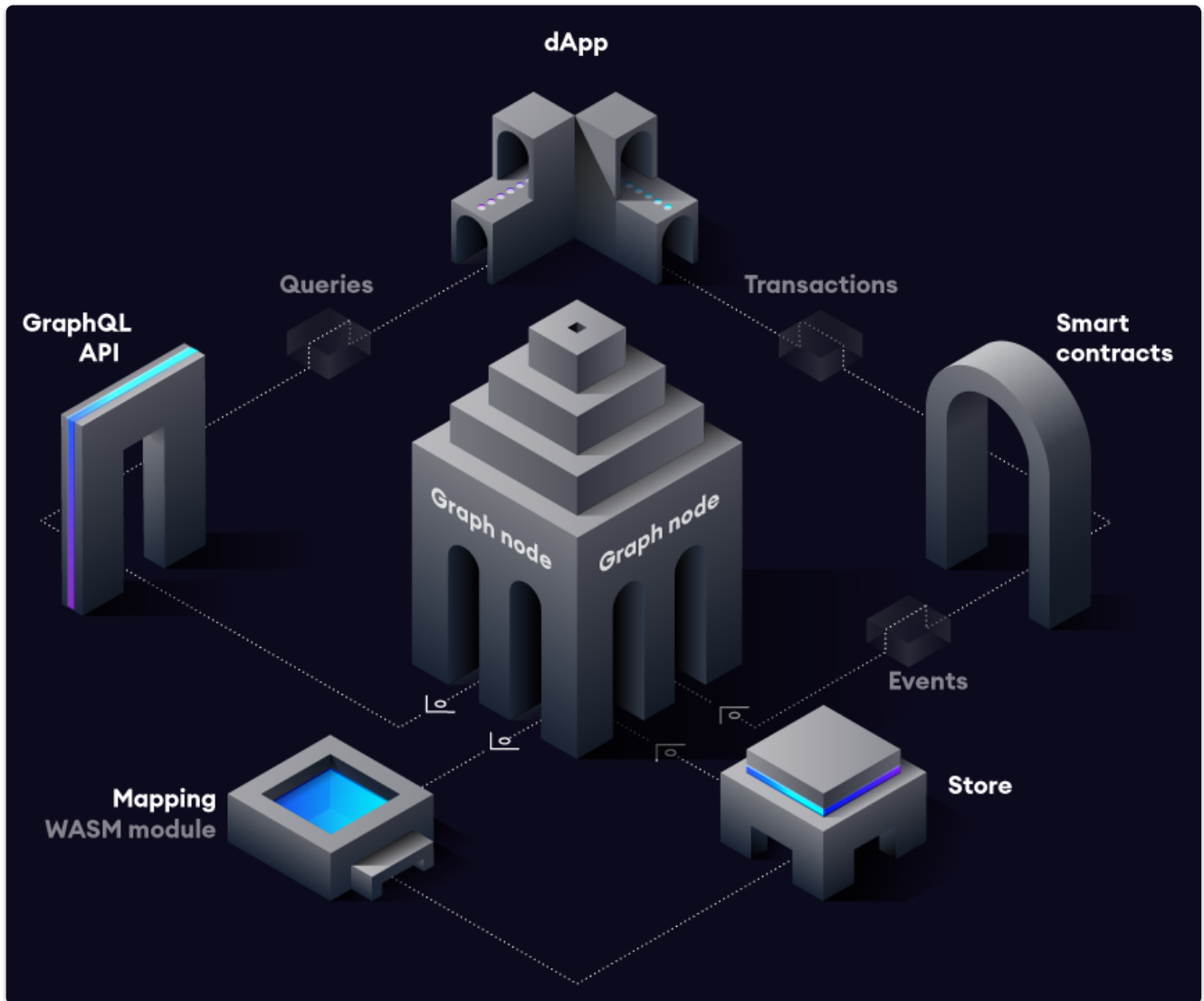
- Comptroller
- Market
- Account
- AccountCToken
- AccountCTokenTransaction
- TransferEvent
- MintEvent
- RedeemEvent
- LiquidationEvent
- BorrowEvent
- RepayEvent
- CTokenTransfer
- UnderlyingTransfer

The Graph learns what and how to index Ethereum data based on subgraph descriptions, known as the subgraph manifest. The subgraph description defines the smart contracts of interest for a subgraph, the events in those contracts to pay attention to, and how to map event data to data that The Graph will store in its database.

Once you have written a subgraph manifest, you use the Graph CLI to store the definition in IPFS and tell the indexer to start indexing data for that subgraph.

This diagram gives more detail about the flow of data once a subgraph manifest has been deployed, dealing with Ethereum transactions:

Data Flow



1. A decentralized application adds data to Ethereum through a transaction on a smart contract.
2. The smart contract emits one or more events while processing the transaction.
3. Graph Node continually scans Ethereum for new blocks and the data for your subgraph they may contain.
4. Graph Node finds Ethereum events for your subgraph in these blocks and runs the mapping handlers you provided. The mapping is a WASM module that creates or updates the data entities that Graph Node stores in response to Ethereum events.
5. The decentralized application queries the Graph Node for data indexed from the blockchain, using the node's GraphQL endpoint. The Graph Node in turn translates the GraphQL queries into queries for its underlying data store in order to fetch this data, making use of the store's indexing capabilities. The decentralized application displays this data in a rich UI for end-users, which they use to issue new transactions on Ethereum. The cycle repeats.

Roles

1. Indexer

Indexers operate nodes and index and serve data through open APIs. They are required to provide a stake and earn rewards for their services.

The indexer's node connects to

- an Ethereum endpoint
- a postgres database
- IPFS

Proof of Indexing

POIs are used in the network to verify that an indexer is indexing the subgraphs they have allocated on. A POI for the first block of the current epoch must be submitted when closing an allocation for that allocation to be eligible for indexing rewards.

Disputes are possible during a certain time window. Network participants that open disputes are called Fishermen, they are required to give a deposit of 10000 GRT.

After the dispute is resolved the indexer may get their stake slashed, and the Fisherman may lose their deposit.

2. Delegator

Delegators delegate their tokens to an indexer, in return they earn a proportion of the query fees.

3. Curator

For end consumers to be able to query a subgraph, the subgraph must first be indexed. Indexing is a process where files, data, and metadata are looked at, cataloged, and then indexed so that results can be found faster. In order for a subgraph's data to be searchable, it needs to be organized.

And so, if Indexers had to guess which subgraphs they should index, there would be a low chance that they would earn robust query fees because they'd have no way of validating which subgraphs are good quality.

Curators assess the web3 ecosystem and signal on the subgraphs that should be indexed by The Graph Network.

Curators make The Graph network efficient and signaling is the process that curators use to let Indexers know that a subgraph is good to index.

4. Developer

Create and publish subgraphs.

Defining a subgraph

We can use Subgraph studio and the CLI to define a subgraph. You need to create an account (connect to a metamask account)

Subgraph studio allows us to create the metadata for the subgraph and gives the steps to use the CLI to define the subgraph.

The CLI will step through the details to allow you create a subgraph from an existing contract (fetching the API from etherscan or falling back to a local ABI file).

This will create some configuration files :

subgraph.yaml

```
specVersion: 0.0.2
schema:
  file: ./schema.graphql
dataSources:
  - kind: ethereum
    name: VolcanoCoin
    network: rinkeby
    source:
      address: "0xe6190da9364067790EF1F402CC78D5CA5DF5D0be"
      abi: VolcanoCoin
    mapping:
      kind: ethereum/events
      apiVersion: 0.0.5
      language: wasm/assemblyscript
      entities:
        - Transfer
        - supplyChanged
      abis:
        - name: VolcanoCoin
          file: ./abis/VolcanoCoin.json
      eventHandlers:
        - event: Transfer(indexed address,uint256)
          handler: handleTransfer
        - event: supplyChanged(uint256)
          handler: handlesupplyChanged
      file: ./src/mapping.ts
```


schema.graphql

```
type ExampleEntity @entity {
  id: ID!
```

```
count: BigInt!  
param0: Bytes! # address  
param1: BigInt! # uint256  
}
```

Querying the Data

Graph Explorer provides endpoints




QUERY SUBGRAPH

Curve


API keys are required to query data from Subgraphs. You can create/manage API keys in Subgraph Studio.


QUERY URL

`https://gateway.thegraph.com/api/[api-key]/subgraphs/id/0x2382ab6c2099474cf424560a370ed1b1fdb65253-0`


Copy Query URL 

For subgraphs you have deployed from Subgraph studio you can also see the endpoint

TEMPORARY QUERY URL 

`https://api.studio.thegraph.com/query/18765/boot2/v1` 

DEPLOYMENT ID

`QmdygtHeeoJ6JWGLtNJR2uHBtr4yz1NAvN6x2Ct4xEkW35` 

SUBGRAPH CREATED

2 days ago

VERSION DEPLOYED

a day ago

UI Integration

[Apollo Client](#) is a Javascript library that allows you to integrate GraphQL queries into your UI.