This page is incomplete and we'd love your help. Edit this page and add anything that you think might be useful to others.

ORACLES

Last edit: @pgrimaud ↗ , April 27, 2021

See contributors

Edit page

On this page >

Oracles are data feeds that connect Ethereum to off-chain, real-world information, so you can query data in your smart contracts. For example, prediction market dapps use oracles to settle payments based on events. A prediction market may ask you to bet your ETH on the next president of the United States. They'll use an oracle to confirm the outcome and pay out to the winners.

PREREQUISITES

Make sure you're familiar with nodes, consensus mechanisms, and smart contract anatomy, specifically events.

WHAT IS AN ORACLE

An oracle is a bridge between the blockchain and the real world. They act as on-chain APIs you can query to get information into your smart contracts. This could be anything from price information to weather reports. Oracles can also be bi-directional, used to "send" data out to the real world.

WHY ARE THEY NEEDED?

With a blockchain like Ethereum you need every node in the network to be able to replay every transaction and end up with the same result, guaranteed. APIs introduce potentially variable data. If you were sending someone an amount of ETH based on an agreed $USD value using a price API, the query would return a different result from one day to the next. Not to mention, the API could be hacked or deprecated. If this happens, the nodes in the network wouldn't be able to agree on Ethereum's current state, effectively breaking consensus.

Oracles solve this problem by posting the data on the blockchain. So any node replaying the transaction will use the

same immutable data that's posted for all to see. To do this, an oracle is typically made up of a smart contract and some off-chain components that can query APIs, then periodically send transactions to update the smart contract's data.

Security

An oracle is only as secure as its data source(s). If a dapp uses Uniswap as an oracle for its ETH/DAI price feed, it is possible for an attacker to move the price on Uniswap in order to manipulate the dapp's understanding of the current price. An example of how to combat this is a feed system↗ like the one used by MakerDAO, which collates price data from a number of external price feeds instead of just relying on a single source.

Architecture

This is an example of a simple Oracle architecture, but there are more ways than this to trigger off-chain computation.

1. Emit a log with your smart contract event
2. An off-chain service has subscribed (usually using something like the JSON-RPC `eth_subscribe` command) to these specific logs.
3. The off-chain service proceeds to do some tasks as defined by the log.
4. The off-chain service responds with the data requested in a secondary transaction to the smart contract.

This is how to get data in a 1 to 1 manner, however to improve security you may want to decentralize how you collect your off-chain data.

The next step might be to have a network of these nodes making these calls to different APIs and sources, and aggregating the data on-chain.

Chainlink Off-Chain Reporting↗ (Chainlink OCR) has improved on this methodology by having the off-chain oracle network communicate with each other, cryptographically sign their responses, aggregate their responses off-chain, and send only 1 transaction on-chain with the result. This way, fewer gas is spent but you still get the guarantee of decentralized data since every node has signed their part of the transaction, making it unchangeable by the node sending the transaction. If the node doesn't transact, the escalation policy kicks in, and the next node sends the transaction.

USAGE

Using services like Chainlink, you can reference decentralized data on-chain, that has already been pulled from the real world and aggregated. Sort of like a public commons, but for decentralized data. You can also build your own modular oracle networks to get any customized data you're looking for. In addition, you can do off-chain computation and send information to the real world as well. Chainlink has infrastructure in place to:

- Get crypto price feeds in your contract↗
- Generate verifiable random numbers (useful for gaming)↗
- Call external APIs↗ – One novel use of this is Checking wBTC reserves↗

This is an example of how to get the latest ETH price in your smart contract using a Chainlink price feed:

Chainlink Data Feeds

Show less Copy

```
pragma solidity ^0.6.7;

import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol";

contract PriceConsumerV3 {

    AggregatorV3Interface internal priceFeed;

    /**
     * Network: Kovan
     * Aggregator: ETH/USD
     * Address: 0x9326BFA02ADD2366b30bacB125260Af641031331
     */
    constructor() public {
        priceFeed = AggregatorV3Interface(0x9326BFA02ADD2366b30bacB125260Af641031331);
    }

    /**
     * Returns the latest price
     */
    function getLatestPrice() public view returns (int) {
        (
            uint80 roundID,
            int price,
            uint startedAt,
            uint timeStamp,
            uint80 answeredInRound
        ) = priceFeed.latestRoundData();
        return price;
    }
}

```

[You can test this in remix with this link ↗]

[View the docs ↗]

Chainlink VRF

Chainlink VRF (Verifiable Random Function) is a provably-fair and verifiable source of randomness designed for smart contracts. Smart contract developers can use Chainlink VRF as a tamper-proof random number generation (RNG) to build reliable smart contracts for any applications which rely on unpredictable outcomes:

- Blockchain games and NFTs.
- Random assignment of duties and resources (e.g. randomly assigning judges to cases).
- Choosing a representative sample for consensus mechanisms.

Random numbers are difficult because blockchains are deterministic.

Working with Chainlink Oracles outside of data feeds follows the request and receive cycle ↗ of working with Chainlink. They use the LINK token to send oracle providers oracle gas for returning responses. The LINK token is specifically designed to work with oracles and are based on the upgraded ERC-677 token, which is backwards compatible with ERC-20. The following code, if deployed on the Kovan testnet will retrieve a cryptographically proven

random number. To make the request, fund the contract with some testnet LINK token that you can get from the Kovan LINK Faucet ↗ .

Show less

```
pragma solidity 0.6.6;

import "@chainlink/contracts/src/v0.6/VRFConsumerBase.sol";

contract RandomNumberConsumer is VRFConsumerBase {

    bytes32 internal keyHash;
    uint256 internal fee;

    uint256 public randomResult;

    /**
     * Constructor inherits VRFConsumerBase
     *
     * Network: Kovan
     * Chainlink VRF Coordinator address: 0xdD3782915140c8f3b190B5D67eAc6dc5760C46E9
     * LINK token address:                0xa36085F69e2889c224210F603D836748e7dC0088
     * Key Hash: 0x6c3699283bda56ad74f6b855546325b68d482e983852a7a82979cc4807b641f4
     */
    constructor()
        VRFConsumerBase(
            0xdD3782915140c8f3b190B5D67eAc6dc5760C46E9, // VRF Coordinator
            0xa36085F69e2889c224210F603D836748e7dC0088  // LINK Token
        ) public
    {
        keyHash = 0x6c3699283bda56ad74f6b855546325b68d482e983852a7a82979cc4807b641f4;
        fee = 0.1 * 10 ** 18; // 0.1 LINK (varies by network)
    }

    /**
     * Requests randomness from a user-provided seed
     */
    function getRandomNumber(uint256 userProvidedSeed) public returns (bytes32 requestId) {
        require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK - fill contract with
faucet");
        return requestRandomness(keyHash, fee, userProvidedSeed);
    }

    /**
     * Callback function used by VRF Coordinator
     */
    function fulfillRandomness(bytes32 requestId, uint256 randomness) internal override {
        randomResult = randomness;
    }
}

```

Chainlink API Call

Chainlink API Calls ↗ are the easiest way to get data from the off-chain world in the traditional way the web works:

API calls. Doing a single instance of this and having only 1 oracle makes it centralized by nature. In order to keep it truly decentralized a smart contract platform would need to use numerous nodes, found in an external data market ↗ .

Deploy the following code in remix on the kovan network to test ↗

This also follows the request and receive cycle of oracles, and needs the contract to be funded with Kovan LINK (the oracle gas) in order to work.

Show less

```
pragma solidity ^0.6.0;

import "@chainlink/contracts/src/v0.6/ChainlinkClient.sol";

contract APIConsumer is ChainlinkClient {

    uint256 public volume;

    address private oracle;
    bytes32 private jobId;
    uint256 private fee;

    /**
     * Network: Kovan
     * Oracle: 0x2f90A6D021db21e1B2A077c5a37B3C7E75D15b7e
     * Job ID: 29fa9aa13bf1468788b7cc4a500a45b8
     * Fee: 0.1 LINK
     */
    constructor() public {
        setPublicChainlinkToken();
        oracle = 0x2f90A6D021db21e1B2A077c5a37B3C7E75D15b7e;
        jobId = "29fa9aa13bf1468788b7cc4a500a45b8";
        fee = 0.1 * 10 ** 18; // 0.1 LINK
    }

    /**
     * Create a Chainlink request to retrieve API response, find the target
     * data, then multiply by 1000000000000000000 (to remove decimal places from data).
     */
    function requestVolumeData() public returns (bytes32 requestId)
    {
        Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this), this.fulfill.selector);

        // Set the URL to perform the GET request on
        request.add("get", "https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD");

        // Set the path to find the desired data in the API response, where the response format is:
        // {"RAW":
        //   {"ETH":
        //    {"USD":
        //     {
        //      "VOLUME24HOUR": xxx.xxx,
        //     }
        //    }
        //   }
        //  }
        request.add("path", "RAW.ETH.USD.VOLUME24HOUR");

        // Multiply the result by 1000000000000000000 to remove decimals
        int timesAmount = 10**18;
        request.addInt("times", timesAmount);

        // Sends the request
```

```
        return sendChainlinkRequestTo(oracle, request, fee);
    }

    /**
     * Receive the response in the form of uint256
     */

    function fulfill(bytes32 _requestId, uint256 _volume) public
recordChainlinkFulfillment(_requestId)
    {
        volume = _volume;
    }
}

```

You can learn more about the applications of chainlink by reading the developers blog ↗ .

ORACLE SERVICES

- Chainlink ↗
- Witnet ↗
- Provable ↗

Build an oracle smart contract

Here's an example oracle contract by Pedro Costa. You can find further annotation in his article: Implementing a Blockchain Oracle on Ethereum ↗ .

Show less 📋 Copy

```
pragma solidity >=0.4.21 <0.6.0;

contract Oracle {
  Request[] requests; //list of requests made to the contract
  uint currentId = 0; //increasing request id
  uint minQuorum = 2; //minimum number of responses to receive before declaring final result
  uint totalOracleCount = 3; // Hardcoded oracle count

  // defines a general api request
  struct Request {
    uint id;                            //request id
    string urlToQuery;                  //API url
    string attributeToFetch;            //json attribute (key) to retrieve in the response
    string agreedValue;                 //value from key
    mapping(uint => string) anwers;     //answers provided by the oracles
    mapping(address => uint) quorum;    //oracles which will query the answer (1=oracle hasn't
voted, 2=oracle has voted)
  }

  //event that triggers oracle outside of the blockchain
  event NewRequest (
    uint id,
    string urlToQuery,
    string attributeToFetch
  );

  //triggered when there's a consensus on the final result
```

```
    //triggered when there's a consensus on the final result
    event UpdatedRequest (
      uint id,
      string urlToQuery,
      string attributeToFetch,
      string agreedValue
    );

    function createRequest (
      string memory _urlToQuery,
      string memory _attributeToFetch
    )
    public
    {
      uint lenght = requests.push(Request(currentId, _urlToQuery, _attributeToFetch, ""));
      Request storage r = requests[lenght-1];

      // Hardcoded oracles address
      r.quorum[address(0x6c2339b46F41a06f09CA0051ddAD54D1e582bA77)] = 1;
      r.quorum[address(0xb5346CF224c02186606e5f89EACC21eC25398077)] = 1;
      r.quorum[address(0xa2997F1CA363D11a0a35bB1Ac0Ff7849bc13e914)] = 1;

      // launch an event to be detected by oracle outside of blockchain
      emit NewRequest (
        currentId,
        _urlToQuery,
        _attributeToFetch
      );

      // increase request id
      currentId++;
    }

    //called by the oracle to record its answer
    function updateRequest (
      uint _id,
      string memory _valueRetrieved
    ) public {

      Request storage currRequest = requests[_id];

      //check if oracle is in the list of trusted oracles
      //and if the oracle hasn't voted yet
      if(currRequest.quorum[address(msg.sender)] == 1){

        //marking that this address has voted
        currRequest.quorum[msg.sender] = 2;

        //iterate through "array" of answers until a position if free and save the retrieved value
        uint tmpI = 0;
        bool found = false;
        while(!found) {
          //find first empty slot
          if(bytes(currRequest.anwers[tmpI]).length == 0){
            found = true;
            currRequest.anwers[tmpI] = _valueRetrieved;
          }
          tmpI++;
        }

        uint currentQuorum = 0;

        //iterate through oracle list and check if enough oracles(minimum quorum)
        //have voted the same answer has the current one
        for(uint i = 0; i < totalOracleCount; i++){
          bytes memory a = bytes(currRequest.anwers[i]);
          bytes memory b = bytes(_valueRetrieved);

          if(keccak256(a) == keccak256(b)){
```

```
            if(keccak256(a) == keccak256(b)){
              currentQuorum++;
              if(currentQuorum >= minQuorum){
                currRequest.agreedValue = _valueRetrieved;
                emit UpdatedRequest (
                  currRequest.id,
                  currRequest.urlToQuery,
                  currRequest.attributeToFetch,
                  currRequest.agreedValue
                );
              }
            }
          }
        }
      }
    }

```

We'd love more documentation on creating an oracle smart contract. If you can help, create a PR!

FURTHER READING

- What is a Blockchain Oracle? ↗ - *Patrick Collins*
- Decentralised Oracles: a comprehensive overview ↗ *–Julien Thevenard*
- Implementing a Blockchain Oracle on Ethereum ↗ *–Pedro Costa*
- Oracles ↗ *–EthHub*

HELP US WITH THIS PAGE

If you're an expert on the topic and want to contribute, edit this page and sprinkle it with your wisdom.

You'll be credited and you'll be helping the Ethereum community!

Use this flexible documentation template ↗

Questions? Ask us in the #content channel on our Discord server ↗

Edit page

Back to top ↑

Yes No

PREVIOUS
ERC-721: NFTs

NEXT
Scaling

Website last updated: June 2, 2021

Cookie Policy

Contact ↗