Oracles

In this chapter we discuss *oracles*, which are systems that can provide external data sources to Ethereum smart contracts. The term "oracle" comes from Greek mythology, where it referred to a person in communication with the gods who could see visions of the future. In the context of blockchains, an oracle is a system that can answer questions that are external to Ethereum.Ideally oracles are systems that are *trustless*, meaning that they do not need to be trusted because they operate on decentralized principles.

Why Oracles Are Needed

A key component of the Ethereum platform is the Ethereum Virtual Machine, with its ability to execute programs and update the state of Ethereum, constrained by consensus rules, on any node in the decentralized network. In order to maintain consensus, EVM execution must be totally deterministic and based only on the shared context of the Ethereum state and signed transactions. This has two particularly important consequences: the first is that there can be no intrinsic source of randomness for the EVM and smart contracts to work with; the second is that extrinsic data can only be introduced as the data payload of a transaction.

Let’s unpack those two consequences further. To understand the prohibition of a true random function in the EVM to provide randomness for smart contracts, consider the effect on attempts to achieve consensus after the execution of such a function: node A would execute the command and store 3 on behalf of the smart contract in its storage, while node B, executing the same smart contract, would store 7 instead. Thus, nodes A and B would come to different conclusions about what the resulting state should be, despite having run exactly the same code in the same context. Indeed, it could be that a different resulting state would be achieved every time that the smart contract is evaluated. As such, there would be no way for the network, with its multitude of nodes running independently around the world, to ever come to a decentralized consensus on what the resulting state should be. In practice, it would get much worse than this example very quickly, because knock-on effects, including ether transfers, would build up exponentially.

Note that pseudorandom functions, such as cryptographically secure hash functions (which are deterministic and therefore can be, and indeed are, part of the EVM), are not enough for many applications. Take a gambling game that simulates coin flips to resolve bet payouts, which needs to randomize heads or tails—a miner can gain an advantage by playing the game and only including their transactions in blocks for which they will win. How do we get around this problem? Since all nodes can agree on the contents of signed transactions, extrinsic information, including sources of randomness, price information, weather forecasts, etc., can be introduced as the data part of transactions sent to the network. However, such data simply cannot be trusted, because it comes from unverifiable sources. As such, we have just deferred the problem. We use oracles to attempt to solve these problems, which we will discuss in detail, in the rest of this chapter.

Oracle Use Cases and Examples

Oracles, ideally, provide a trustless (or at least near-trustless) way of getting extrinsic (i.e., "real-world" or off-chain) information, such as the results of football games, the price of gold, or truly random numbers, onto the Ethereum platform for smart contracts to use. They can also be used to

relay data securely to DApp frontends directly. Oracles can therefore be thought of as a mechanism for bridging the gap between the off-chain world and smart contracts. Allowing smart contracts to enforce contractual relationships based on real-world events and data broadens their scope dramatically. However, this can also introduce external risks to Ethereum's security model. Consider a "smart will" contract that distributes assets when a person dies. This is something frequently discussed in the smart contract space, and highlights the risks of a trusted oracle. If the inheritance amount controlled by such a contract is high enough, the incentive to hack the oracle and trigger distribution of assets *before* the owner dies is very high.

Note that some oracles provide data that is particular to a specific private data source, such as academic certificates or government IDs. The source of such data, such as a university or government department, is fully trusted, and the truth of the data is subjective (truth is only determined by appeal to the authority of the source). Such data cannot therefore be provided trustlessly—i.e., without trusting a source—as there is no independently verifiable objective truth. As such, we include these data sources in our definition of what counts as "oracles" because they also provide a data bridge for smart contracts. The data they provide generally takes the form of attestations, such as passports or records of achievement. Attestations will become a big part of the success of blockchain platforms in the future, particularly in relation to the related issues of verifying identity or reputation, so it is important to explore how they can be served by blockchain platforms.

Some more examples of data that might be provided by oracles include:

- Random numbers/entropy from physical sources such as quantum/thermal processes: e.g., to fairly select a winner in a lottery smart contract
- Parametric triggers indexed to natural hazards: e.g., triggering of catastrophe bond smart contracts, such as Richter scale measurements for an earthquake bond
- Exchange rate data: e.g., for accurate pegging of cryptocurrencies to fiat currency
- Capital markets data: e.g., pricing baskets of tokenized assets/securities
- Benchmark reference data: e.g., incorporating interest rates into smart financial derivatives
- Static/pseudostatic data: security identifiers, country codes, currency codes, etc.
- Time and interval data: for event triggers grounded in precise time measurements
- Weather data: e.g., insurance premium calculations based on weather forecasts
- Political events: for prediction market resolution
- Sporting events: for prediction market resolution and fantasy sports contracts
- Geolocation data: e.g., as used in supply chain tracking
- Damage verification: for insurance contracts
- Events occurring on other blockchains: interoperability functions
- Ether market price: e.g., for fiat gas price oracles
- Flight statistics: e.g., as used by groups and clubs for flight ticket pooling

In the following sections, we will examine some of the ways oracles can be implemented, including basic oracle patterns, computation oracles, decentralized oracles, and oracle client implementations

in Solidity.

Oracle Design Patterns

All oracles provide a few key functions, by definition. These include the ability to:

- Collect data from an off-chain source.
- Transfer the data on-chain with a signed message.
- Make the data available by putting it in a smart contract's storage.

Once the data is available in a smart contract's storage, it can be accessed by other smart contracts via message calls that invoke a "retrieve" function of the oracle's smart contract; it can also be accessed by Ethereum nodes or network-enabled clients directly by "looking into" the oracle's storage.

The three main ways to set up an oracle can be categorized as *request–response*, *publish-subscribe*, and *immediate-read*.

Starting with the simplest, *immediate-read* oracles are those that provide data that is only needed for an immediate decision, like "What is the address for *ethereumbook.info*?" or "Is this person over 18?" Those wishing to query this kind of data tend to do so on a "just-in-time" basis; the lookup is done when the information is needed and possibly never again. Examples of such oracles include those that hold data about or issued by organizations, such as academic certificates, dial codes, institutional memberships, airport identifiers, self-sovereign IDs, etc. This type of oracle stores data once in its contract storage, whence any other smart contract can look it up using a request call to the oracle contract. It may be updated. The data in the oracle's storage is also available for direct lookup by blockchain-enabled (i.e., Ethereum client–connected) applications without having to go through the palaver and incurring the gas costs of issuing a transaction. A shop wanting to check the age of a customer wishing to purchase alcohol could use an oracle in this way. This type of oracle is attractive to an organization or company that might otherwise have to run and maintain servers to answer such data requests. Note that the data stored by the oracle is likely not to be the raw data that the oracle is serving, e.g., for efficiency or privacy reasons. A university might set up an oracle for the certificates of academic achievement of past students. However, storing the full details of the certificates (which could run to pages of courses taken and grades achieved) would be excessive. Instead, a hash of the certificate is sufficient. Likewise, a government might wish to put citizen IDs onto the Ethereum platform, where clearly the details included need to be kept private. Again, hashing the data (more carefully, in Merkle trees with salts) and only storing the root hash in the smart contract's storage would be an efficient way to organize such a service.

The next setup is *publish–subscribe*, where an oracle that effectively provides a broadcast service for data that is expected to change (perhaps both regularly and frequently) is either polled by a smart contract on-chain, or watched by an off-chain daemon for updates. This category has a pattern similar to RSS feeds, WebSub, and the like, where the oracle is updated with new information and a flag signals that new data is available to those who consider themselves "subscribed." Interested parties must either poll the oracle to check whether the latest information has changed, or listen for updates to oracle contracts and act when they occur. Examples include price feeds, weather information, economic or social statistics, traffic data, etc. Polling is very inefficient in the world of web servers, but not so in the peer-to-peer context of blockchain

platforms: Ethereum clients have to keep up with all state changes, including changes to contract storage, so polling for data changes is a local call to a synced client. Ethereum event logs make it particularly easy for applications to look out for oracle updates, and so this pattern can in some ways even be considered a "push" service. However, if the polling is done from a smart contract, which might be necessary for some decentralized applications (e.g., where activation incentives are not possible), then significant gas expenditure may be incurred.

The *request–response* category is the most complicated: this is where the data space is too huge to be stored in a smart contract and users are expected to only need a small part of the overall dataset at a time. It is also an applicable model for data provider businesses. In practical terms, such an oracle might be implemented as a system of on-chain smart contracts and off-chain infrastructure used to monitor requests and retrieve and return data. A request for data from a decentralized application would typically be an asynchronous process involving a number of steps. In this pattern, firstly, an EOA transacts with a decentralized application, resulting in an interaction with a function defined in the oracle smart contract. This function initiates the request to the oracle, with the associated arguments detailing the data requested in addition to supplementary information that might include callback functions and scheduling parameters. Once this transaction has been validated, the oracle request can be observed as an EVM event emitted by the oracle contract, or as a state change; the arguments can be retrieved and used to perform the actual query of the off-chain data source. The oracle may also require payment for processing the request, gas payment for the callback, and permissions to access the requested data. Finally, the resulting data is signed by the oracle owner, attesting to the validity of the data at a given time, and delivered in a transaction to the decentralized application that made the request—either directly or via the oracle contract. Depending on the scheduling parameters, the oracle may broadcast further transactions updating the data at regular intervals (e.g., end-of-day pricing information).

The steps for a request–response oracle may be summarized as follows:

1. Receive a query from a DApp.
2. Parse the query.
3. Check that payment and data access permissions are provided.
4. Retrieve relevant data from an off-chain source (and encrypt it if necessary).
5. Sign the transaction(s) with the data included.
6. Broadcast the transaction(s) to the network.
7. Schedule any further necessary transactions, such as notifications, etc.

A range of other schemes are also possible; for example, data can be requested from and returned directly by an EOA, removing the need for an oracle smart contract. Similarly, the request and response could be made to and from an Internet of Things–enabled hardware sensor. Therefore, oracles can be human, software, or hardware.

The request–response pattern described here is commonly seen in client–server architectures. While this is a useful messaging pattern that allows applications to have a two-way conversation, it is perhaps inappropriate under certain conditions. For example, a smart bond requiring an interest rate from an oracle might have to request the data on a daily basis under a request–response pattern in order to ensure the rate is always correct. Given that interest rates change infrequently, a

publish–subscribe pattern may be more appropriate here—especially when taking into consideration Ethereum's limited bandwidth.

Publish–subscribe is a pattern where publishers (in this context, oracles) do not send messages directly to receivers, but instead categorize published messages into distinct classes. Subscribers are able to express an interest in one or more classes and retrieve only those messages that are of interest. Under such a pattern, an oracle might write the interest rate to its own internal storage each time it changes. Multiple subscribed DApps can simply read it from the oracle contract, thereby reducing the impact on network bandwidth while minimizing storage costs.

In a broadcast or multicast pattern, an oracle would post all messages to a channel and subscribing contracts would listen to the channel under a variety of subscription modes. For example, an oracle might publish messages to a cryptocurrency exchange rate channel. A subscribing smart contract could request the full content of the channel if it required the time series for, e.g., a moving average calculation; another might require only the latest rate for a spot price calculation. A broadcast pattern is appropriate where the oracle does not need to know the identity of the subscribing contract.

Data Authentication

If we assume that the source of data being queried by a DApp is both authoritative and trustworthy (a not insignificant assumption), an outstanding question remains: given that the oracle and the request–response mechanism may be operated by distinct entities, how are we able trust this mechanism? There is a distinct possibility that data may be tampered with in transit, so it is critical that off-chain methods are able to attest to the returned data's integrity. Two common approaches to data authentication are *authenticity proofs* and *trusted execution environments* (TEEs).

Authenticity proofs are cryptographic guarantees that data has not been tampered with. Based on a variety of attestation techniques (e.g., digitally signed proofs), they effectively shift the trust from the data carrier to the attestor (i.e., the provider of the attestation). By verifying the authenticity proof on-chain, smart contracts are able to verify the integrity of the data before operating upon it. Oraclize is an example of an oracle service leveraging a variety of authenticity proofs.One such proof that is currently available for data queries from the Ethereum main network is the TLSNotary proof. TLSNotary proofs allow a client to provide evidence to a third party that HTTPS web traffic occurred between the client and a server. While HTTPS is itself secure, it doesn't support data signing. As a result, TLSNotary proofs rely on TLSNotary (via PageSigner) signatures. TLSNotary proofs leverage the Transport Layer Security (TLS) protocol, enabling the TLS master key, which signs the data after it has been accessed, to be split between three parties: the server (the oracle), an auditee (Oraclize), and an auditor. Oraclize uses an Amazon Web Services (AWS) virtual machine instance as the auditor, which can be verified as having been unmodified since instantiation. This AWS instance stores the TLSNotary secret, allowing it to provide honesty proofs. Although it offers higher assurances against data tampering than a pure request–response mechanism, this approach does require the assumption that Amazon itself will not tamper with the VM instance.

Town Crier is an authenticated data feed oracle system based on the TEE approach; such methods utilize hardware-based secure enclaves to ensure data integrity. Town Crier uses Intel'sSoftware Guard eXtensions (SGX) to ensure that responses from HTTPS queries can be verified as authentic.

SGX provides guarantees of integrity, ensuring that applications running within an enclave are protected by the CPU against tampering by any other process. It also provides confidentiality, ensuring that an application's state is opaque to other processes when running within the enclave. And finally, SGX allows attestation, by generating a digitally signed proof that an application—securely identified by a hash of its build—is actually running within an enclave. By verifying this digital signature, it is possible for a decentralized application to prove that a Town Crier instance is running securely within an SGX enclave. This, in turn, proves that the instance has not been tampered with and that the data emitted by Town Crier is therefore authentic. The confidentiality property additionally enables Town Crier to handle private data by allowing data queries to be encrypted using the Town Crier instance's public key. Operating an oracle's query/response mechanism within an enclave such as SGX effectively allows us to think of it as running securely on trusted third-party hardware, ensuring that the requested data is returned untampered with (assuming that we trust Intel/SGX).

Computation Oracles

So far, we have only discussed oracles in the context of requesting and delivering data. However, oracles can also be used to perform arbitrary computation, a function that can be especially useful given Ethereum's inherent block gas limit and comparatively expensive computation costs. Rather than just relaying the results of a query, computation oracles can be used to perform computation on a set of inputs and return a calculated result that may have been infeasible to calculate on-chain. For example, one might use a computation oracle to perform a computationally intensive regression calculation in order to estimate the yield of a bond contract.

If you are willing to trust a centralized but auditable service, you can go again to Oraclize. They provide a service that allows decentralized applications to request the output of a computation performed in a sandboxed AWS virtual machine. The AWS instance creates an executable container from a user-configured Dockerfile packed in an archive that is uploaded to the Inter-Planetary File System (IPFS; see [data_storage_sec]). On request, Oraclize retrieves this archive using its hash and then initializes and executes the Docker container on AWS, passing any arguments that are provided to the application as environment variables. The containerized application performs the calculation, subject to a time constraint, and writes the result to standard output, where it can be retrieved by Oraclize and returned to the decentralized application. Oraclize currently offers this service on an auditable t2.micro AWS instance, so if the computation is of some nontrivial value, it is possible to check that the correct Docker container was executed. Nonetheless, this is not a truly decentralized solution.

The concept of a 'cryptlet' as a standard for verifiable oracle truths has been formalized as part of Microsoft's wider ESC Framework. Cryptlets execute within an encrypted capsule that abstracts away the infrastructure, such as I/O, and has the CryptoDelegate attached so incoming and outgoing messages are signed, validated, and proven automatically. Cryptlets support distributed transactions so that contract logic can take on complex multistep, multiblockchain, and external system transactions in an ACID manner. This allows developers to create portable, isolated, and private resolutions of the truth for use in smart contracts. Cryptlets follow the format shown here:

```
public class SampleContractCryptlet : Cryptlet
  {
        public SampleContractCryptlet(Guid id, Guid bindingId, string name,
```

```
            string address, IContainerServices hostContainer, bool contract)
            : base(id, bindingId, name, address, hostContainer, contract)
        {
            MessageApi = new CryptletMessageApi(GetType().FullName,
                new SampleContractConstructor())
```

For a more decentralized solution, we can turn to TrueBit, which offers a solution for scalable and verifiable off-chain computation. They use a system of solvers and verifiers who are incentivized to perform computations and verification of those computations, respectively. Should a solution be challenged, an iterative verification process on subsets of the computation is performed on-chain—a kind of 'verification game'. The game proceeds through a series of rounds, each recursively checking a smaller and smaller subset of the computation. The game eventually reaches a final round, where the challenge is sufficiently trivial such that the judges—Ethereum miners—can make a final ruling on whether the challenge was met, on-chain. In effect, TrueBit is an implementation of a computation market, allowing decentralized applications to pay for verifiable computation to be performed outside of the network, but relying on Ethereum to enforce the rules of the verification game. In theory, this enables trustless smart contracts to securely perform any computation task.

A broad range of applications exist for systems like TrueBit, ranging from machine learning to verification of proof of work. An example of the latter is the Doge–Ethereum bridge, which uses TrueBit to verify Dogecoin's proof of work (Scrypt), which is a memory-hard and computationally intensive function that cannot be computed within the Ethereum block gas limit. By performing this verification on TrueBit, it has been possible to securely verify Dogecoin transactions within a smart contract on Ethereum's Rinkeby testnet.

Decentralized Oracles

While centralized data or computation oracles suffice for many applications, they represent single points of failure in the Ethereum network. A number of schemes have been proposed around the idea of decentralized oracles as a means of ensuring data availability and the creation of a network of individual data providers with an on-chain data aggregation system.

ChainLink has proposed a decentralized oracle network consisting of three key smart contracts—a reputation contract, an order-matching contract, and an aggregation contract—and an off-chain registry of data providers. The reputation contract is used to keep track of data providers' performance. Scores in the reputation contract are used to populate the off-chain registry. The order-matching contract selects bids from oracles using the reputation contract. It then finalizes a service-level agreement, which includes query parameters and the number of oracles required. This means that the purchaser needn't transact with the individual oracles directly. The aggregation contract collects responses (submitted using a commit–reveal scheme) from multiple oracles, calculates the final collective result of the query, and finally feeds the results back into the reputation contract.

One of the main challenges with such a decentralized approach is the formulation of the aggregation function. ChainLink proposes calculating a weighted response, allowing a validity score to be reported for each oracle response. Detecting an 'invalid' score here is nontrivial, since it relies on the premise that outlying data points, measured by deviations from responses provided by

peers, are incorrect. Calculating a validity score based on the location of an oracle response among a distribution of responses risks penalizing correct answers over average ones. Therefore, ChainLink offers a standard set of aggregation contracts, but also allows customized aggregation contracts to be specified.

A related idea is the SchellingCoin protocol. Here, multiple participants report values and the median is taken as the "correct" answer. Reporters are required to provide a deposit that is redistributed in favor of values that are closer to the median, therefore incentivizing the reporting of values that are similar to others. A common value, also known as the Schelling point, which respondents might consider as the natural and obvious target around which to coordinate is expected to be close to the actual value.

Jason Teutsch of TrueBit recently proposed a new design for a decentralized off-chain data availability oracle. This design leverages a dedicated proof-of-work blockchain that is able to correctly report on whether or not registered data is available during a given epoch. Miners attempt to download, store, and propagate all currently registered data, thereby guaranteeing data is available locally. While such a system is expensive in the sense that every mining node stores and propagates all registered data, the system allows storage to be reused by releasing data after the registration period ends.

Oracle Client Interfaces in Solidity

Using Oraclize to update the ETH/USD exchange rate from an external source is a Solidity example demonstrating how Oraclize can be used to continuously poll for the ETH/USD price from an API and store the result in a usable manner.

Example 1. Using Oraclize to update the ETH/USD exchange rate from an external source

```
/*
   ETH/USD price ticker leveraging CryptoCompare API

   This contract keeps in storage an updated ETH/USD price,
   which is updated every 10 minutes.
 */

pragma solidity ^0.4.1;
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

/*
   "oraclize_" prepended methods indicate inheritance from "usingOraclize"
 */
contract EthUsdPriceTicker is usingOraclize {

    uint public ethUsd;

    event newOraclizeQuery(string description);
    event newCallbackResult(string result);

    function EthUsdPriceTicker() payable {
```

```
        // signals TLSN proof generation and storage on IPFS
        oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);

        // requests query
        queryTicker();
    }

    function __callback(bytes32 _queryId, string _result, bytes _proof) public {
        if (msg.sender != oraclize_cbAddress()) throw;
        newCallbackResult(_result);

        /*
         * Parse the result string into an unsigned integer for on-chain use.
         * Uses inherited "parseInt" helper from "usingOraclize", allowing for
         * a string result such as "123.45" to be converted to uint 12345.
         */
        ethUsd = parseInt(_result, 2);

        // called from callback since we're polling the price
        queryTicker();
    }

    function queryTicker() external payable {
        if (oraclize_getPrice("URL") > this.balance) {
            newOraclizeQuery("Oraclize query was NOT sent, please add some ETH
                to cover for the query fee");
        } else {
            newOraclizeQuery("Oraclize query was sent, standing by for the
                answer...");

            // query params are (delay in seconds, datasource type,
            // datasource argument)
            // specifies JSONPath, to fetch specific portion of JSON API result
            oraclize_query(60 * 10, "URL",
                "json(https://min-api.cryptocompare.com/data/price?\
                fsym=ETH&tsyms=USD,EUR,GBP).USD");
        }
    }
}
```

To integrate with Oraclize, the contract EthUsdPriceTicker must be a child of `usingOraclize`; the usingOraclize contract is defined in the *oraclizeAPI* file. The data request is made using the oraclize_query function, which is inherited from the usingOraclize contract. This is an overloaded function that expects at least two arguments:

- The supported data source to use, such as URL, WolframAlpha, IPFS, or computation
- The argument for the given data source, which may include the use of JSON or XML parsing helpers

The price query is performed in the queryTicker function. In order to perform the query, Oraclize requires the payment of a small fee in ether, covering the gas cost for processing the result and transmitting it to the __callback function and an accompanying surcharge for the service. This amount is dependent on the data source and, where specified, the type of authenticity proof that is required. Once the data has been retrieved, the __callback function is called by an Oraclize-controlled account permissioned to do the callback; it passes in the response value and a unique queryId argument, which, for example, can be used to handle and track multiple pending callbacks from Oraclize.

Financial data provider Thomson Reuters also provides an oracle service for Ethereum, called BlockOne IQ, allowing market and reference data to be requested by smart contracts running on private or permissioned networks. Contract calling the BlockOne IQ service for market data shows the interface for the oracle, and a client contract that will make the request.

Example 2. Contract calling the BlockOne IQ service for market data

```
pragma solidity ^0.4.11;

contract Oracle {
    uint256 public divisor;
    function initRequest(
       uint256 queryType, function(uint256) external onSuccess,
       function(uint256
    ) external onFailure) public returns (uint256 id);
    function addArgumentToRequestUint(uint256 id, bytes32 name, uint256 arg)
public;
    function addArgumentToRequestString(uint256 id, bytes32 name, bytes32 arg)
        public;
    function executeRequest(uint256 id) public;
    function getResponseUint(uint256 id, bytes32 name) public constant
        returns(uint256);
    function getResponseString(uint256 id, bytes32 name) public constant
        returns(bytes32);
    function getResponseError(uint256 id) public constant returns(bytes32);
    function deleteResponse(uint256 id) public constant;
}

contract OracleB1IQClient {

    Oracle private oracle;
    event LogError(bytes32 description);

    function OracleB1IQClient(address addr) external payable {
        oracle = Oracle(addr);
        getIntraday("IBM", now);
    }

    function getIntraday(bytes32 ric, uint256 timestamp) public {
        uint256 id = oracle.initRequest(0, this.handleSuccess,
this.handleFailure);
```

```
        oracle.addArgumentToRequestString(id, "symbol", ric);
        oracle.addArgumentToRequestUint(id, "timestamp", timestamp);
        oracle.executeRequest(id);
    }

    function handleSuccess(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 ric = oracle.getResponseString(id, "symbol");
        uint256 open = oracle.getResponseUint(id, "open");
        uint256 high = oracle.getResponseUint(id, "high");
        uint256 low = oracle.getResponseUint(id, "low");
        uint256 close = oracle.getResponseUint(id, "close");
        uint256 bid = oracle.getResponseUint(id, "bid");
        uint256 ask = oracle.getResponseUint(id, "ask");
        uint256 timestamp = oracle.getResponseUint(id, "timestamp");
        oracle.deleteResponse(id);
        // Do something with the price data
    }

    function handleFailure(uint256 id) public {
        assert(msg.sender == address(oracle));
        bytes32 error = oracle.getResponseError(id);
        oracle.deleteResponse(id);
        emit LogError(error);
    }

}
```

The data request is initiated using the initRequest function, which allows the query type (in this example, a request for an intraday price) to be specified, in addition to two callback functions. This returns a uint256 identifier that can then be used to provide additional arguments. The addArgumentToRequestString function is used to specify the Reuters Instrument Code (RIC), here for IBM stock, and `addArgumentToRequestUint` allows the timestamp to be specified. Now, passing in an alias for block.timestamp will retrieve the current price for IBM. The request is then executed by the executeRequest function. Once the request has been processed, the oracle contract will call the onSuccess callback function with the query identifier, allowing the resulting data to be retrieved; in the event of retrieval failure, the `onFailure` callback will return an error code instead. The available fields that can be retrieved on success include open, high, low, close (OHLC), and bid/ask prices.

Conclusions

As you can see, oracles provide a crucial service to smart contracts: they bring external facts to contract execution. With that, of course, oracles also introduce a significant risk—if they are trusted sources and can be compromised, they can result in compromised execution of the smart contracts they feed.

Generally, when considering the use of an oracle be very careful about the *trust model*. If you assume the oracle can be trusted, you may be undermining the security of your smart contract by

exposing it to potentially false inputs. That said, oracles can be very useful if the security assumptions are carefully considered.

Decentralized oracles can resolve some of these concerns and offer Ethereum smart contracts trustless external data. Choose carefully and you can start exploring the bridge between Ethereum and the "real world" that oracles offer.