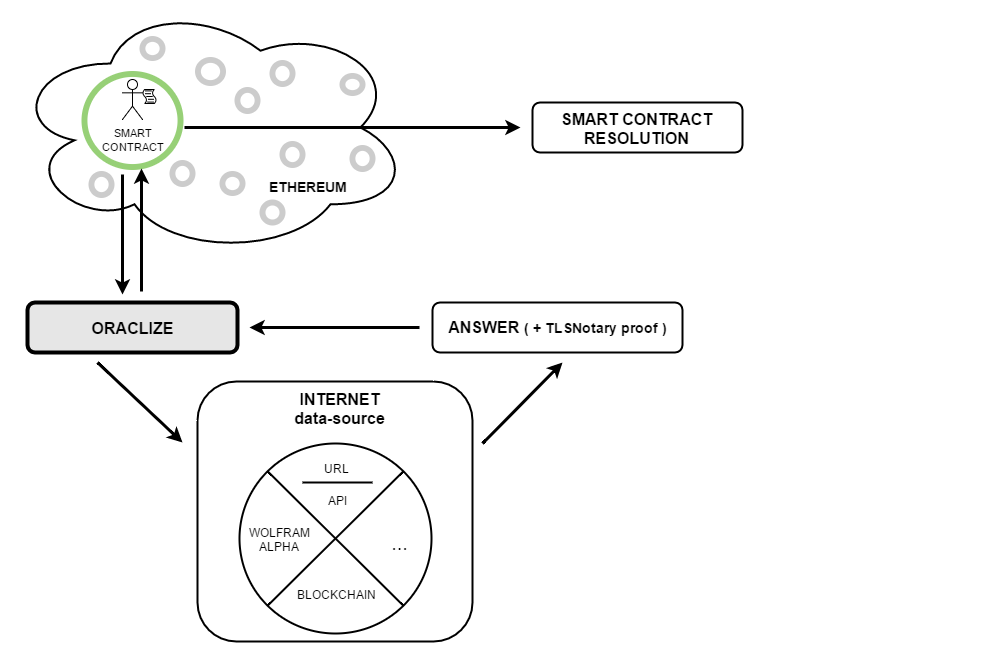
Oracle service builds the bridge between blockchain and the real world, and we are going to introduce an oracle service called [**Oraclize**](http://www.oraclize.it/)**.**

The following image describes the underlying mechanism of the Oraclize.



It uses the **TLSNotary** to make its service trust-worthy, and pre-fetch the real world data for all the nodes within the network to achieve consensus, which cannot be easily done with Ethereum smart contract.

# **Need for Oraclize**

The need for **Oraclize** arise from the fact that blockchain applications, such as Bitcoin scripts and smart contracts cannot access and fetch directly the data they require: price feeds for assets and financial applications; weather-related information for peer-to-peer insurance; random number generation for gambling.

But to rely on a new trusted intermediary, the oracle in this case, it would be betraying the security and reduced-trust model of blockchain applications: which is what makes them interesting and useful in first place.

One solution is to accept data inputs from more than one untrusted or partially trusted party and then execute the data-dependent action only after a number of them have provided the same answer or an answer within some constrains. This type of system can be considered a decentralized oracle system.

Unfortunately, this approach has severe **limitations**:

* It requires **a predefined standard** on data format
* It is inherently **inefficient**: all the parties participating will require a fee and, for every request, it will take time before reaching a sufficient number of answers.

The solution developed by Oraclize is instead to demonstrate that the data fetched from the original data-source is **genuine** and **untampered**. This is accomplished by accompanying the returned data together with a document called **authenticity** proof. The authenticity proofs can build upon different technologies such **as auditable virtual machines** and **Trusted Execution Environments.**

Oraclize engine can be easily integrated with both private and public instances of different blockchain protocols.

# **Oraclize Engine**

The Oraclize Engine powers the service for both blockchain-based and non-blockchain-based application. Internally replicates an **"If This Then That"** logical model. This means that it will execute a given set of instructions if some other given conditions are met. This flexibility enables the engine to be leveraged in many different ways and contexts.

A valid request for data to Oraclize, done via the native blockchain integration or via the HTTP API, should specify the following arguments:

* A data source type
* A query
* Optionally, an authenticity proof type

# **Data Source Types**

A data source is a trusted provider of data. It can be a website or web API such **as Reuters, Weather.com, BBC.com**, or a secure application running on a **hardware-enforced Trusted Execution Environment (TEE)** or an **auditable, locked-down virtual machine instance running in a cloud provider**. Oraclize currently offers the following types of native data sources:

* **URL**: enables the access to any webpage or **HTTP** API endpoint
* **WolframAlpha**: enables native access to WolframAlpha computational engine
* **IPFS**: provides access to any content stored on an IPFS file
* **random**: provides untampered random bytes coming from a secure application running on a Ledger Nano S.
* **computation**: provides the result of arbitrary computation

Additionally, there also some meta data source such as:

* **nested**: enables the combination of different types of data source or multiple requests using the same data source, and it returns a unique result
* **identity**: it returns the query
* **decrypt**: it decrypts a string encrypted to the Oraclize private key

# **Query**

A query is an array of parameters which needs to be evaluated in order to complete a specific data source type request: query: [ parameter\_1, parameters\_2, ...];

The first parameter is the main argument and it is usually mandatory. For example, in the case of the URL Data Source Type, the first argument is the expected URL where the resource resides. If only the first argument is present, then the URL Data Source assumes that an HTTP GET was requested. The second parameters, which it is optional, should contain the data payload of the HTTP POST request.

# **Parsing Helpers**

Oraclize offers the following parser helpers:

* JSON Parser
* XML Parser
* HTML Parser
* Binary Parser

**\*Binary helper must be used with the slice option and only raw binary inputs are accepted**

# **Authenticity Proofs**

Oraclize offers the following Authenticity Proofs:

* **TLS Notary Proof:** The TLSNotary Proof leverages a feature of the **TLS 1.0 and 1.1** protocols which enables the splitting of the TLS master key between three parties**: the server, an auditee and an auditor**. In this scheme**, Oraclize is the auditee** while a locked-down AWS instance of a specially-designed**, open-source Amazon Machine Image acts as the auditor**. The TLSNotary protocol is an open-source technology, developed and used by the **PageSigner** project.
* **Android Proof:** The Android Proof is a result of some of Oraclize's internal R&D work. It leverages software remote attestation technology developed by **Google**, called **SafetyNet**, to validate that a given Android application is running on a safe, non-rooted physical device, connected to Oraclize's infrastructure. It also remotely validates the application code hash, enabling authentication of the application running on the device.
* **Ledger Proof:** [**Ledger**](https://www.ledger.co/) is a French company, leader in the production of hardware-enforced cryptocurrency wallets. They have controller and operating system called **BOLOS**. BOLOS exposes a set of **kernel-level API** which can complete useful operations such as cryptographic ones or attestation. The Ledger Proof leverages both the **code attesting and the device attesting features** to attest to any third-party that the applications developed by Oraclize are running in a TEE of a true Ledger device.

## Cost for Different Types of Proofs: -



## Data Sources Vs Different Proofs: -



# **Integration with Ethereum: -**

The interaction between Oraclize and an Ethereum smart contract is **asynchronous**. Any request for data is composed of two steps:

1. Firstly, in the most common case, **a transaction executing a function of a smart contract** is **broadcasted** by a user. The function contains **a special instruction which manifest to Oraclize, who is constantly monitoring the Ethereum blockchain for such instruction**, a request for data.
2. Secondly, according to the parameters of such request, **Oraclize will fetch or compute a result, build, sign and broadcast the transaction carrying the result**. In the default configuration, such transaction will execute the**\_\_callback**function which should be placed in the smart contract by its developer:

The generation of an authenticity proof is **optional** and it is **a contract-wide setting** which must be configured by the smart contract developer before the request for data is initiated. Oraclize always recommends the use of authenticity proofs for production deployments.

# **Implementation**

## Points to Remember: -

* The contract should be a child of the contract **usingOraclize**
* The contract usingOraclize is defined in the **oraclizeAPI** file, which can be fetched from the dedicated Oraclize Github repository.

## Simple Query: -

A request for data is called query. The oraclize\_query is a function, inhered from the parent usingOraclize contract, which expects at least two arguments:

* A **data-source** such as URL, WolframAlpha, IPFS, 'Swarm' and others listed here
* The **argument for the given data-source**. For examples: the full URL, which may include the use of JSON or XML parsing helpers or a WolframAlpha formula or an IPFS multihash

// This code example will ask Oraclize to send as soon as possible

// a transaction with the primary result (as a string) of the given

// formula ("random number between 0 and 100") fetched from the

// data-source "WolframAlpha".

oraclize\_query ("WolframAlpha", "random number between 0 and 100");

oraclize\_query ("URL", "https://api.kraken.com/0/public/Ticker?pair=ETHXBT")

oraclize\_query ("URL",

"json(https://www.therocktrading.com/api/ticker/BTCEUR).result.0.last")

oraclize\_query ("IPFS", "QmdEJwJG1T9rzHvBD8i69HHuJaRgXRKEQCP7Bh1BVttZbU")

// The URL datasource also supports a supplement argument, useful for creating HTTP POST requests.

// If that argument is a valid JSON string, it will be automatically sent as JSON.

oraclize\_query ("URL", "json(https://shapeshift.io/sendamount).success.deposit",

'{"pair":"eth\_btc","amount":"1","withdrawal":"1AAcCo21EUc1jbocjssSQDzLna9Vem2UN5"}')

## Scheduled Query: -

The execution of a query can **be scheduled in a future date**. The function oraclize\_query accepts as a parameter the **delay in seconds from the current time or the timestamp** in the future as first argument. I**n order for the future timestamp to be accepted by Oraclize it must be within 60 days of the current UTC time** in the case of the absolute timestamp choice, or in the case of a relative time elapse, the elapsed seconds must equate to no more than 60 days.

// Relative time: get the result from the given URL 60 seconds from now

oraclize\_query (60, "URL",

"json(https://api.kraken.com/0/public/Ticker?pair=ETHXBT).result.XETHXXBT.c.0")

// Absolute time: get the result from the given datasource at the specified UTC timestamp in the future

oraclize\_query (scheduled\_arrivaltime+3\*3600,

"WolframAlpha", strConcat ("flight ", flight\_number, " landed"));

## Recursive Queries: -

Smart contracts using Oraclize can be effectively autonomous by implementing a new call to Oraclize into their **\_\_callback** method. This can be useful for implementing **periodic updates of some on-chain reference data**, as with price feeds, or to periodically check for some off-chain conditions.

pragma solidity ^0.4.11;

import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract ExampleContract is usingOraclize {

string public ETHUSD;

event LogConstructorInitiated(string nextStep);

event LogPriceUpdated(string price);

event LogNewOraclizeQuery(string description);

function ExampleContract() payable {

LogConstructorInitiated("Constructor was initiated. Call 'updatePrice()' to send the Oraclize Query.");

}

function \_\_callback(bytes32 myid, string result) {

if (msg.sender != oraclize\_cbAddress()) revert();

ETHUSD = result;

LogPriceUpdated(result);

updatePrice();

}

function updatePrice() payable {

if (oraclize\_getPrice("URL") > this.balance) {

LogNewOraclizeQuery("Oraclize query was NOT sent, please add some ETH to cover for the query fee");

} else {

LogNewOraclizeQuery("Oraclize query was sent, standing by for the answer..");

oraclize\_query(60, "URL", "json(https://api.pro.coinbase.com/products/ETH-USD/ticker).price");

}

}

## Query ID: -

Every time the function oraclize\_query is called, it returns a **unique ID,** hereby referred to as **queryId**, which **depends from the number of previous requests and the address of smart contract**. The queryId identifies a specific query done to Oraclize and it is returned to the contract as a parameter of the callback transaction.

Example from the above code sample being modified:

function \_\_callback(bytes32 myid, string result) {

if (!validIds[myid]) revert();

if (msg.sender != oraclize\_cbAddress()) revert();

ETHUSD = result;

LogPriceUpdated(result);

delete validIds[myid];

updatePrice();

}

function updatePrice() payable {

if (oraclize\_getPrice("URL") > this.balance) {

LogNewOraclizeQuery("Oraclize query was NOT sent, please add some ETH to cover for the query fee");

} else {

LogNewOraclizeQuery("Oraclize query was sent, standing by for the answer..");

bytes32 queryId =

oraclize\_query(60, "URL", "json(https://api.pro.coinbase.com/products/ETH-USD/ticker).price");

validIds[queryId] = true;

}

}

## Custom Gas Limit and Gas Price: -

If no settings are specified, Oraclize will use the default values of **200,000 gas** and **20 GWei**. A different value for the Oraclize callback gas can be passed as the argument **\_gasLimit** to the **oraclize\_query** function as shown in the following example:

function updatePrice() payable {

if (oraclize\_getPrice("URL") > this.balance) {

LogNewOraclizeQuery("Oraclize query was NOT sent, please add some ETH to cover for the query fee");

} else {

LogNewOraclizeQuery("Oraclize query was sent, standing by for the answer..");

bytes32 queryId =

oraclize\_query(60, "URL", "json(https://api.pro.coinbase.com/products/ETH-USD/ticker).price", 500000);

validIds[queryId] = true;

}

}

The gas price of the callback transaction can be set by calling the **oraclize\_setCustomGasPrice** function, either in the **constructor**, or in a **separate function.**

Smart contract developers should **estimate** correctly and **minimize** the cost of their **\_\_callback** method, as **any unspent gas will be returned to Oraclize and no refund is available.**

The following is the ExampleContract modified to specify a custom gas price of 4 Gwei and a custom gas limit for the callback transaction.

function ExampleContract() payable {

oraclize\_setCustomGasPrice(4000000000);

LogConstructorInitiated("Constructor was initiated. Call 'updatePrice()' to send the Oraclize Query.");

}

## Authenticity Proofs: -

When a smart contract requests for an authenticity proof, it must **define** a different callback function with the following arguments: **function \_\_callback(bytes32 queryId, string result, bytes proof).**

The **oraclize\_setProof** function expects the following format: **oraclize\_setProof(proofType\_ | proofStorage\_** ).

Both proofType and proofStorage are byte constants defined in usingOraclize:

Available parameters for proofTypes are:

* **proofType\_NONE:** the **default** value of any smart contracts
* **proofType\_TLSNotary**
* **proofType\_Android**
* **proofType\_Native**
* **proofType\_Ledger**

While for proofStorage:

* **proofStorage\_IPFS**

Example:

function ExampleContract() payable {

oraclize\_setCustomGasPrice(4000000000);

oraclize\_setProof(proofType\_TLSNotary | proofStorage\_IPFS);

LogConstructorInitiated("Constructor was initiated. Call 'updatePrice()' to send the Oraclize Query.");

}

# Encrypted Queries

Oraclize therefore offers the possibility of encrypting the parameters contained in a query to Oraclize's public key**: 044992e9473b7d90ca54d2886c7addd14a61109af202f1c95e218b0c99eb060c7134c4ae46345d0383ac996185762f04997d6fd6c393c86e4325c469741e64eca9.** **Only** **Oraclize** will then be able to decrypt the request using its **paired private key**.

To encrypt the query, Oraclize provides a **CLI tool**. Alternatively, the **CLI command** to encrypt an arbitrary string of text is:

**python encrypted\_queries\_tools.py -e -p 044992e9473b7d90ca54d2886c7addd14a61109af202f1c95e218b0c99eb060c7134c4ae46345d0383ac996185762f04997d6fd6c393c86e4325c469741e64eca9 *"YOUR QUERY"***

// In this example, the entire first argument of an oraclize\_query has been encrypted.

// The actual string encrypted is: json(https://poloniex.com/public?command=returnTicker).BTC\_ETH.last

oraclize\_query ("URL","AzK149Vj4z65WphbBPiuWQ2PStTINeVp5sS9PSwqZi8NsjQy6jJLH765qQu3U/

bZPNeEB/bYZJYBivwmmREXTGjmKJk/62ikcO6mIMQfB5jBVVUOqzzZ/A8ecWR2nOLv0CKkkkFzBYp2sW1H

31GI+SQzWV9q64WdqZsAa4gXqHb6jmLkVFjOGI0JvrA/Zh6T5lyeLPSmaslI");

This will encrypt the query with the default Oraclize public key. The **encrypted string** can then be used as an **argument** for an Oraclize query. The encryption method is also available for **POST** requests.

// This is the query that we want to encrypt

oraclize\_query("URL","json(https://api.postcodes.io/postcodes).status",

'{"postcodes" : ["OX49 5NU", "M32 0JG", "NE30 1DP"]}')

// Finally we add all the encrypted text

// to the oraclize\_query (in the right order)

oraclize\_query ("BEIGVzv6fJcFiYQNZF8ArHnvNMAsAWBz8Zwl0YCsy4K/RJTN8ERHfBWtSfYHt+

uegdD1wtXTkP30sTW+3xR3w/un1i3caSO0Rfa+wmIMmNHt4aOS","BNKdFtmfmazLLR/bfey4mP8

v/R5zCIUK7obcUrF2d6CWUMvKKUorQqYZNu1YfRZsGlp/F96CAQhSGomJC7oJa3PktwoW5J1Oti/

y2v4+b5+vN8yLIj1trS7p1l341Jf66AjaxnoFPplwLqE=", "BF5u1td9ugoacDabyfVzoTxPBxG

NtmXuGV7AFcO1GLmXkXIKlBcAcelvaTKIbmaA6lXwZCJCSeWDHJOirHiEl1LtR8lCt+1ISttWuvp

J6sPx3Y/QxTajYzxZfQb6nCGkv+8cczX0PrqKKwOn/Elf9kpQQCXeMglunT09H2B4HfRs7uuI");

To protect the plaintext queries, an **Elliptic Curve Integrated Encryption** Scheme is used.

# **Computation Data Source**

## Passing Arguments to the Package: -

Arguments can be passed to the package by **adding parameters to the query array**. They will be accessible from within the **Docker instances as environmental parameters**.

Currently the API supports up to **5 inline arguments**, including the IPFS Hash:

***oraclize\_query ("computation”, ["QmZRjkL4U72XFXTY8MVcchpZciHAwnTem51AApSj6Z2byR", \_firstOperand, \_secondOperand, \_thirdOperand, \_fourthOperand]);***

## Passing more than 5 Arguments: -

In case we need to pass more arguments, we will need to send a manually set **dynamic string/bytes array**, for example:

***string[] memory myArgs = new string[](6);***

***myArgs[0] = "MYIPFSHASH";***

***...***

***myArgs[5] = "LAST ARG";***

The query would then look like this: ***oraclize\_query ("computation", myArgs)***

## Passing Encrypted Arguments: -

Encrypted arguments can be passed using the nested and the decrypt meta data sources.

oraclize\_query ("nested", "[computation] ['QmaqMYPnmSHEgoWRMP3WSrUYsPWKjT85C81PgJa2SXBs8u', \

'Example of decrypted string', '${[decrypt] BOYnQstP700X10I+WWNUVVNZEmal+rZ0GD1CgcW5P5wUSFKr2QoIwHLvkHfQR5e4Bfakq0CIviJnjkfKFD+ZJzzxcaFUQITDZJxsRLtKuxvAuh6IccUJ+jDF/znTH+8x8EE1Tt9SY7RvqtVao2vxm4CxIWq1vk4=}', 'Hello there!']");

}

# **Proof Shield**

The **ProofShield** enables smart contracts to **verify on-chain the authenticity proofs** provided by Oraclize, this ensures that the authenticity of the data received is verified before going ahead and using the data.

To enable the ProofShield it is enough to set it via the **oraclize\_setProof** function like you see in the following code:

oraclize\_setProof (proofType\_Android\_v2 | proofShield\_Ledger);

Once the ProofShield is enabled, the received proof **will not be the raw Authenticity Proof**, but the **ProofShield proof** instead: some functions are provided so that the ProofShield proof can be verified on-chain. In order to verify it, we need to call from within the **\_\_callback**method the function **oraclize\_proofShield\_proofVerify\_\_returnCode (queryId, result, proof)** and ensure that it **returns 0.** A code example follows,

function \_\_callback(bytes32 queryId, string result, bytes proof) {

if (msg.sender != oraclize\_cbAddress()) revert();

if (oraclize\_proofShield\_proofVerify\_\_returnCode(queryId, result, proof) != 0) {

// the proof verification has failed, do we need to take any action here? (depends on the use case)

} else {

// the proof verification has passed

// now that we know that the random number was safely generated, let's use it..

LogNewAuthenticatedResult(result);

}

}

## Note: -

The ProofShield is still **EXPERIMENTAL**, and is **NOT** to be used in production. A production-ready version will follow in the future.

# **Storage and Delivery of Authenticity Proofs**

The authenticity proofs may be relatively **large** files, of up to a few kilobytes. Delivering such proofs directly within the result of the data payload in an Ethereum transaction can get **quite expensive**, in terms of EVM execution costs, and may even be impossible for larger data.

Moreover, **Oraclize strives to be blockchain agnostic**, enabling the proof to be used even on Bitcoin and other blockchains. Therefore, the proof is **uploaded and saved to IPFS**.

# **Summary of the Oraclize Workflow**

* From our smart contract, we call **oraclize\_query ("URL", "YourHTTPurl")** (as an example)**.**
* This **triggers an event**. Oraclize **watches** these events and **performs HTTPS call to YourHttpURL.**
* If we demand for this, Oraclize can use **TLSNotary** to generate a proof of the call. The proof is **stored** on IPFS.
* Oraclize **receives** the results, and **delivers** it to our smart contract via the **\_\_callback(result**) function. We have to provide a proper \_\_callback() function in our smart contract.
* We have to **process the results** inside your \_\_callback() function.

# **A Sample Smart Contract to fetch Temperature of a City using Oraclize via the OpenWeather API**

pragma solidity ^0.4.11;

import "github.com/oraclize/ethereum-api/oraclizeAPI\_0.4.sol";

contract Test is usingOraclize {

string public TEMPERATURE;

event LogConstructorInitiated (string nextStep);

event LogPriceUpdated (string price);

event LogNewOraclizeQuery (string description);

constructor() public payable {

emit LogConstructorInitiated ("Constructor was initiated. Call 'getWeather()' to send the Oraclize Query.");

}

function \_\_callback (bytes32 myid, string result) public {

if (msg.sender != oraclize\_cbAddress()) revert();

TEMPERATURE = strConcat(result,"°C");

emit LogPriceUpdated(result);

}

function getWeather() public payable {

if (oraclize\_getPrice("URL") > address(this).balance) {

emit LogNewOraclizeQuery ("Oraclize query was NOT sent, please add some ETH to cover for the query fee");

} else {

emit LogNewOraclizeQuery ("Oraclize query was sent, standing by for the answer."); oraclize\_query("URL","json(https://api.openweathermap.org/data/2.5/weather?id=1275004&units=metric&APPID=84a070758462174005f30a6dc80267d4).main.temp");

}

}

}

# **Reference**

* <http://docs.oraclize.it/>
* <https://github.com/johnhckuo/Oraclize-Tutorial>
* <https://www.reddit.com/r/ethereum/comments/6cf2df/quick_question_how_do_oracles_work_in_ethereum_if/>