Advanced - API Calls

Calling APIs from Smart Contracts

> 👍 **Prerequisites**
>
> This tutorial requires basic knowledge about Ethereum, smart contracts, and the Chainlink Request & Receive cycle. If you're unfamiliar with those concepts, follow the Beginners or Intermediates tutorials.

By the end of the tutorial, you should know the following:

- How to request data from a public API in a smart contract
- Understand what Core Adapters and External Adapters are and how Oracle Jobs use them
- Be able to find the Oracle Jobs and Adapters for your contract
- How to request data from an Oracle Job

1. Requesting API Data

1a. Request & Receive Recap

The request and receive cycle describes how a smart contract requests data from an oracle and receives the response in a separate transaction. If you need a refresher, check out the Basic Request Model.

In the Intermediates tutorial, we request randomness from a VRF oracle, then await the response. The fulfilment function is already given to us from the `VRFConsumerBase` contract, so oracles already know where to send the response to. However, with API calls, our contract *defines* which function it wants to receive the response to.

However, before we go into the implementation, let's first understand how Oracle jobs can get data on-chain.

1b. Initiators

Initiators are what kick off a job inside an Oracle. In the case of a Request and Receive job, the RunLog initiator watches the blockchain for when a smart contract makes a request. Once it catches a request, it initiates the job. This runs the adapters (both core and external) that the job is configured to run, eventually returning the response to the contract that made the request.

1c. Core Adapters

Each oracle job has a configured set of tasks it needs to carry out when it is run. These tasks are defined by what Adapters they support. For example: if a job needs to make a GET request to an API, find a specific unsigned integer field in a JSON response, then submit that back to the requesting contract, it would need a job with the following Core Adapters:

- HttpGet - Call the API
- JsonParse - Parse the JSON and retrieve the desired data
- EthUint256 - Convert the data to Ethereum compatible data type (uint256)
- EthTx - Submit the transaction to the chain, completing the cycle.

Let's walk through a real example, where we retrieve 24 volume of the ETH/USD pair from the cryptocompare API.

Core Adapters Example

1. HttpGet - Calls the API and returns the body of an HTTP GET result for ETH/USD pair. Example:


```
{"RAW":
  {"ETH":
    {"USD":
      {
        ...,
        "VOLUMEDAYTO":953806939.7194247,
        "VOLUME24HOUR":703946.0675653099,
        "VOLUME24HOURTO":1265826345.488568
        ...,
      }
    }
  }
}
```


2. JsonParse - walks a specified `path` (`"RAW.ETH.USD.VOLUME24HOUR"`) and returns the value found at that result. Example: `703946.0675653099`

3. Multiply - parses the input into a float and multiplies it by the 10^18. Example: `703946067565309900000000`

4. EthUint256 - formats the input into an integer and then converts it into Solidity's `uint256` format. Example: `0xc618a1e4`

5. EthTx - takes the given input, places it into the data field of the transaction, signs a transaction, and broadcasts it to the network. Example: transaction result

Important: Some core adapters accept parameters to be passed to them to inform them how to run. For example: JsonParse accepts a `path` parameter which informs the adapter where to find the data in the JSON object.

Let's see what this looks like in a contract.

Contract Example

```
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
     * Chainlink - 0x2f90A6D021db21e1B2A077c5a37B3C7E75D15b7e
     * Chainlink - 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), th

        // Set the URL to perform the GET request on
        request.add("get", "https://min-api.cryptocompare.com/data/pricemultifull?fsyms=E

        // Set the path to find the desired data in the API response, where the response
        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 recordChainlinkFulfillme
    {
        volume = _volume;
    }
}
```

Deploy this contract using Remix ↗

Let's walk through what's happening here:

1. Constructor - Setup the contract with the Oracle address, Job ID, and LINK fee that the oracle charges for the job
2. `requestVolumeData` - This builds and sends a request, which includes the fulfilment functions selector, to the oracle. Notice how it adds the `get`, `path` and `times` parameters. These are read by the Adapters in the job to perform the tasks correctly. `get` is used by HttpGet, `path` is used by JsonParse and `times` is used by Multiply.
3. `fulfill` - Where the result is sent once the Oracle job is complete

> 📘 **LINK Required**
>
> Note, the calling contract should own enough LINK to pay the specified fee (by default 0.1 LINK). You can use this tutorial to fund your contract.

This was an example of a basic HTTP GET request. However, it requires defining the API URL directly in the smart contract. This can, in fact, be extracted and configured on the Job level inside the Oracle.

1d. External Adapters

We split Adapters into two subcategories:

- Core Adapters - These are what we described earlier and come built-in to each node. (examples: HttpGet, EthUint256, etc)
- External Adapters - These are custom adapters built by node operators and community members, which perform specific tasks like calling a particular endpoint with a specific set of parameters (like authentication secrets that shouldn't be publicly visible smart contracts).

Here are some examples of external adapters:

1. Markets data: AlphaChain
2. Real-world events: SportsData, COVID Tracker
3. Social media proofs: MUBC Retweet Verifier
4. Cryptocurrency aggregators: Coingecko, CoinAPI

All of these can be found on Chainlink Market.

If all the parameters are defined within the Oracle job, the only thing a smart contract needs to define to consume it is:

- JobId
- Oracle address
- LINK fee
- Fulfillment function

This makes for a much more succinct smart contract, where the `requestVolumeData` function from the code example above would look more like this:

```
function requestVolumeData() public returns (bytes32 requestId) {
    Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this), this.f
```

```
    // Extra parameters don't need to be defined here because they are already defined in

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

2. Exercise: Construct your own Contract

Now that we know how core adapters and external adapters are used to construct jobs and how smart contracts can use jobs to make requests let's put that to use!

Head to Make an Existing Job Request to see how a smart contract can get any city's temperature using an existing oracle job found on Chainlink Market, without having to specify the URL inside the contract.

Then, using your knowledge of external adapters, find a different adapter on the market, and create another contract that consumes that data. Let us know the cool things you come up with in our discord!

3. Further Reading

- Blog: Connect a Smart Contract to the Twitter API
- Blog: Connect a Tesla Vehicle API to a Smart Contract
- Blog: OAuth and API Authentication in Smart Contracts