

# Call Multiple Data Sources

This tutorial shows you how to make multiple API calls from your smart contract to a Decentralized Oracle Network. After [CCR](#) completes offchain computation and aggregation, the DON returns the asset price to your smart contract. This example returns the BTC/USD price.

This guide assumes that you know how to build HTTP requests and how to use secrets. Read the [API query parameters](#) and [API use secrets](#) guides before you follow the example in this document. To build a decentralized asset price, send a request to the DON to fetch the price from many different API providers. Then, calculate the median price. The API providers in this example are:

- [CoinMarket](#)
- [CoinGecko](#)
- [CoinPaprika](#)

caution

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

note

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the [terms](#) can result in suspension or termination of your Chainlink Functions account.

## Prerequisites

note

You might skip these prerequisites if you have followed one of these [guides](#). You can check your subscription details (including the balance in LINK) in the [Chainlink Functions Subscription Manager](#). If your subscription runs out of LINK, follow the [Fund a Subscription](#) guide.

## Set up your environment

You must provide the private key from a testnet wallet to run the examples in this documentation. Install a Web3 wallet, configure [node.js](#), clone the [smartcontractkit/smart-contract-examples](#) repository, and configure `a.env.encfile` with the required environment variables.

Install and configure your Web3 wallet for Polygon Mumbai:

1. [Install Dero](#) so you can compile and simulate your Functions source code on your local machine.
2. [Install the MetaMask wallet](#) or other Ethereum Web3 wallet.
3. Set the network for your wallet to the Polygon Mumbai testnet. If you need to add Mumbai to your wallet, you can find the chain ID and the LINK token contract address on the [LINK Token Contracts](#) page.
4. [Polygon Mumbai testnet and LINK token contract](#)
5. Request testnet MATIC from the [Polygon Faucet](#).
6. Request testnet LINK from [faucets.chain.link/mumbai](#).

Install the required frameworks and dependencies:

1. [Install the latest release of Node.js 20](#). Optionally, you can use the [nvm package](#) to switch between Node.js versions with `nvm use 20`.

Note: To ensure you are running the correct version in a terminal, type `node -v`.

`node -v` node-vv20.9.0 2. In a terminal, clone the [smart-contract-examples](#) repository and change directories. This example repository imports the [Chainlink Functions Toolkit NPM package](#). You can import this package to your own projects to enable them to work with Chainlink Functions.

`git clone https://github.com/smartcontractkit/smart-contract-examples.git` & `cd ./smart-contract-examples/functions-examples/` 3. `Run npm install` to install the dependencies.

`npm install` 4. For higher security, the examples repository encrypts your environment variables at rest.

1. Set an encryption password for your environment variables.

`npmx env-enc set -pw` 2. `Run npmx env-enc set` to configure `a.env.encfile` with the basic variables that you need to send your requests to the Polygon Mumbai network.

- POLYGON\_MUMBAI\_RPC\_URL: Set a URL for the Polygon Mumbai testnet. You can sign up for a personal endpoint from [Alchemy Infura](#), or another node provider service.
- PRIVATE\_KEY: Find the private key for your testnet wallet. If you use MetaMask, follow the instructions to [export a Private Key](#). Note: Your private key is needed to sign any transactions you make such as making requests.

`npmx env-enc set`

## Configure your onchain resources

After you configure your local environment, configure some onchain resources to process your requests, receive the responses, and pay for the work done by the DON.

## Deploy a Functions consumer contract on Polygon Mumbai

1. [Open the FunctionsConsumerExample.sol contract](#) in Remix.

[Open in Remix](#) What is Remix? 2. Compile the contract. 3. Open MetaMask and select the Polygon Mumbai network. 4. In Remix under the Deploy & Run Transaction tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Polygon Mumbai. 5. Under the Deploy section, fill in the router address for your specific blockchain. You can find both of these addresses on the [Supported Networks](#) page. For Polygon Mumbai, the router address is `0x6E2dc0F9DB014aE19888F539E59285D2Ea04244C`. 6. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Polygon Mumbai. 7. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy the contract address.

## Create a subscription

Follow the [Managing Functions Subscriptions](#) guide to accept the Chainlink Functions Terms of Service (ToS), create a subscription, fund it, then add your consumer contract address to it.

You can find the Chainlink Functions Subscription Manager at [functions.chain.link](#).

## Tutorial

This tutorial is configured to get the median BTC/USD price from multiple data sources. For a detailed explanation of the code example, read the [Examine the code](#) section.

You can locate the scripts used in this tutorial in the [examples/8-multiple-apis](#) directory.

1. Make sure your subscription has enough LINK to pay for your requests. Also, you must maintain a minimum balance to upload encrypted secrets to the DON (Read the [minimum balance for uploading encrypted secrets](#) section to learn more). You can check your subscription details (including the balance in LINK) in the [Chainlink Functions Subscription Manager](#). If your subscription runs out of LINK, follow the [Fund a Subscription](#) guide. This guide recommends maintaining at least 2 LINK within your subscription.
2. Get a free API key from [CoinMarketCap](#) and note your API key.
3. `Run npmx env-enc set` to add an encrypted COINMARKETCAP\_API\_KEY to your `a.env.encfile`.

`npmx env-enc set`

To run the example:

1. Open the `file request.js`, which is located in the `8-multiple-apis` folder.
2. Replace the consumer contract address and the subscription ID with your own values.

`const consumerAddress = "0x8dFf78B7EE3128D00E90611FBED20A71397064D9"// REPLACE this with your Functions consumer address`  
`const subscriptionId = 3// REPLACE this with your subscription ID` 3. Make a request:

- CoinMarketCap:
- ```
curl -X'GET' "https://pro-api.coinmarketcap.com/v1/cryptocurrency/quotes/latest?id=1&convert=USD" -H'accept: application/json' -H'X-CMC_PRO_API_KEY: REPLACE_WITH_YOUR_API_KEY' *
CoinGecko:
```

```
curl-X'GET'https://api.coingecko.com/api/v3/simple/price?vs_currencies=USD&ids=bitcoin'-H'accept: application/json' * Coinpaprika:
```

```
curl-X'GET'https://api.coinpaprika.com/v1/tickers/btc-bitcoin'-H'accept: application/json'
```

The prices are respectively located at:

- CoinMarketCap:data,1,quote,USD,price
- CoinGecko:bitcoin,usd
- Coinpaprika:quotes,USD,price

The main steps of the scripts are:

- Construct the HTTP objects `coinMarketCapRequest`, `coinGeckoRequest`, and `coinPaprikaRequest` using `Functions.makeHttpRequest`. The values for `coinMarketCapCoinId`, `coinGeckoCoinId`, and `coinPaprikaCoinId` are fetched from the args.
- Make the HTTP calls.
- Read the asset price from each response.
- Calculate the median of all the prices.
- Return the result as a [buffer](#) using the `Functions.encodeUint256` helper function. Because solidity doesn't support decimals, multiply the result by 100 and round the result to the nearest integer. Note: Read this [article](#) if you are new to Javascript Buffers and want to understand why they are important.

## request.js

This explanation focuses on the [request.js](#) script and shows how to use the [Chainlink Functions NPM package](#) in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- [path](#) and [fs](#): Used to read the [source file](#).
- [ethers](#): Ethers.js library, enables the script to interact with the blockchain.
- [@chainlink/functions-toolkit](#): Chainlink Functions NPM package. All its utilities are documented in the [NPM README](#).
- [@chainlink/env-enc](#): A tool for loading and storing encrypted environment variables. Read the [official documentation](#) to learn more.
- [./abi/functionsClient.json](#): The abi of the contract your script will interact with. Note: The script was tested with this [FunctionsConsumerExample contract](#).

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

```
const consumerAddress = "0x8dFf78B7EE3128D00E90611FBeD20A71397064D9"// REPLACE this with your Functions consumer address
const subscriptionId = 3// REPLACE this with your subscription ID
```

The primary function that the script executes is `makeRequestMumbai`. This function can be broken into six main parts:

1. Definition of necessary identifiers:
2. `routerAddress`: Chainlink Functions router address on Polygon Mumbai.
3. `donId`: Identifier of the DON that will fulfill your requests on Polygon Mumbai.
4. `gatewayUrls`: The secrets endpoint URL to which you will upload the encrypted secrets.
5. `explorerUrl`: Block explorer url of Polygon Mumbai.
6. `source`: The source code must be a string object. That's why we use `fs.readFileSync` to read `source.js` and then `callToString()` to get the content as a string object.
7. `args`: During the execution of your function, These arguments are passed to the source code. The `args` value is `["1", "bitcoin", "btc-bitcoin"]`. These arguments are BTC IDs at CoinMarketCap, CoinGecko, and Coinpaprika. You can adapt args to fetch other asset prices.
8. `secrets`: The secrets object that will be encrypted.
9. `slotIdNumber`: Slot ID at the DON where to upload the encrypted secrets.
10. `expirationTimeMinutes`: Expiration time in minutes of the encrypted secrets.
11. `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
12. Initialization of `ethersigner` and `provider` objects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.
13. Simulating your request in a local sandbox environment:
14. Use `simulateScript` from the Chainlink Functions NPM package.
15. Read the `response` of the simulation. If successful, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).
16. Estimating the costs:
17. Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost` function.
18. The response is returned in Juels (1 LINK = 10\*\*18 Juels). Use the `ethers.utils.formatEther` utility function to convert the output to LINK.
19. Encrypt the secrets, then upload the encrypted secrets to the DON. This is done in two steps:
20. Initialize a `SecretsManager` instance from the Functions NPM package, then call the `encryptSecrets` function.
21. Call the `uploadEncryptedSecretsToDON` function of the `SecretsManager` instance. This function returns an object containing a `success` boolean as long as a `version`, the secret version on the DON storage. Note: When making the request, you must pass the slot ID and version to tell the DON where to fetch the encrypted secrets.
22. Making a Chainlink Functions request:
23. Initialize your functions consumer contract using the contract address, abi, and ethers signer.
24. Call the `sendRequest` function of your consumer contract.
25. Waiting for the response:
26. Initialize a `ResponseListener` from the Functions NPM package and then call the `listenForResponseFromTransaction` function to wait for a response. By default, this function waits for five minutes.
27. Upon reception of the response, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).