

Encode request data offchain

This tutorial shows you how 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 theBTC/USDprice.

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](#)

Read the[Call Multiple Data Sources](#) tutorial before you follow the steps in this example. This tutorial uses the same example but with a slightly different process:

- Instead of sending the request data (source code, encrypted secrets reference, and arguments) in the request, you will first encode it offchain and then send the encoded request. Encoding the request offchain from your front end or a server rather than onchain from your smart contract. This helps save gas.

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 Deno](#) 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.

npx env-enc set-pw 2. Run npx 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.

npx 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/9-send-cbdir](#) 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 npx env-enc set to add an encrypted COINMARKETCAP_API_KEY to your .env.encfile.

npx env-enc set

To run the example:

1. Open the `filerequest.js`, which is located in the `9-send-cbor` folder.
2. Replace the consumer contract address and the subscription ID with your own values.

constconsumerAddress="0x8dF78B7EE3128D00E90611FBED20A71397064D9"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID 3. Make a request:

`nodeexamples/9-send-cbor/request.is` The script runs your function in a sandbox environment before making an onchain transaction.

\$ node examples/9-send-cbor/request.js secp256k1 unavailable, reverting to browser version Start simulation... Performing simulation with the following versions: deno 1.36.3 (release, aarch64-apple-darwin) v8 11.6.189.12 typescript 5.1.6

[illegible]

Estimate request costs... Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256)) Fulfillment cost estimated to 0.0000000000000215 LINK

Make request... Upload encrypted secret to gateways <https://01.functions-gateway.testnet.chain.link/user>. StorageSlotId 0. Expiration in minutes: 15

✓ Secrets uploaded properly to gateways <https://01.functions-gateway.testnet.chain.link/user/> Gateways response: { version: 1693899379, success: true }

Functions request sent! Transaction hash 0x310c93ba1c9515a8e1dc4308e198c36f5f915410fc67c13f80ffb29a43dcbe60. Waiting for a response... See your request in the explorer <https://mumbai.polygonscan.com/tx/0x310c93ba1c9515a8e1dc4308e198c36f5f915410fc67c13f80ffb29a43dcbe60>

[illegible]

✓ Decoded response to uint256: 2571336nThe output of the example gives you the following information:

- Your request is first run on a sandbox environment to ensure it is correctly configured.
- The fulfillment costs are estimated before making the request.
- The encrypted secrets were uploaded to the secrets endpoint <https://01.functions.gateway.testnet.chain.link/user>.
- Your request was successfully sent to Chainlink Functions. The transaction in this example is [9x310c93ba1c9515a8e1dc4308e198c36f5f915410cf67c13f80ffb29a43dcb60](#) and the request ID is 0x74b5b88db49dad155d1cd9b0da47cb140caf1f824cf09a88bc46132c21d41d20.
- The DON successfully fulfilled your request. The total cost was: 0.000039224977086446 LINK.
- The consumer contract received a response in bytes with a value of 0x00273c48. Decoding it offchain to Uint256 gives you a result: 2571336. The median RUC price is 25713.36 USD.

Examine the code

FunctionsConsumerExample.sol

```
SPDX-License-Identifier: MIT
pragma solidity 0.8.19;
import (FunctionsClient) from "chainlink/contracts/src/v0.8/functions/dev/v1_0_0/FunctionsClient.sol";
import (ConfirmedOwner) from "chainlink/contracts/src/v0.8/shared/access/ConfirmedOwner.sol";

// THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY. * THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE. * DO NOT USE THIS CODE IN PRODUCTION.

// contract FunctionsConsumerExample is FunctionsClient, ConfirmedOwner {
//     using FunctionsRequest for FunctionsRequest;
//     bytes32 public lastRequestId;
//     bytes public lastResponse;

//     /**
//      * @notice Send a simple request *
//      * @param source JavaScript source code *
//      * @param encryptedSecretsUrls Encrypted URLs where to fetch user secrets *
//      * @param donHostedSecretsSlotID Don hosted secrets slotID *
//      * @param donHostedSecretsVersion Don hosted secrets version *
//      * @param args List of arguments accessible from within the source code *
//      * @param bytesArgs Array of bytes arguments, represented as hex strings *
//      * @param subscriptionID Billing ID
//      *
//      * @return requestID The ID of the sent request
//      */
//     function sendRequest(string memory source, bytes memory encryptedSecretsUrls, uint8 donHostedSecretsSlotID, uint64 donHostedSecretsVersion, string[] memory args, bytes[] memory bytesArgs, uint64 subscriptionID) public {
//         FunctionsRequest request = FunctionsRequest.initForInlineJavaScript(source);
//         if (encryptedSecretsUrls.length > 0) request.addSecretsReference(encryptedSecretsUrls);
//         if (donHostedSecretsVersion > 0) request.addDONHostedSecrets(donHostedSecretsSlotID, donHostedSecretsVersion);
//         if (args.length > 0) request.setArgs(args);
//         if (bytesArgs.length > 0) request.setBytesArgs(bytesArgs);
//         s_lastRequestId = request.sendRequest();

//         /**
//          * @notice Send a pre-encoded CBOR request *
//          * @param request CBOR-encoded request data *
//          * @param subscriptionID Billing ID *
//          * @param gasLimit The maximum amount of gas the request can consume *
//          * @param donID ID of the job to be invoked *
//          * @return requestID The ID of the sent request
//          */
//         function sendRequestCBOR(bytes memory request, uint64 subscriptionID, uint32 gasLimit, bytes32 donID) external onlyOwner returns (bytes32 requestID) {
//             s_lastRequestId = request.sendRequest(subscriptionID, gasLimit, donID);
//             return s_lastRequestId;
//         }

//         /**
//          * @param response Aggregated response from the user code *
//          * @param err Aggregated error from the user code or from the execution pipeline *
//          * Either response or error parameter will be set, but never both
//          */
//         function fulfillRequest(bytes32 requestID, bytes memory response, bytes memory err) internal override {
//             if (s_lastRequestId != requestID) {
//                 revert UnexpectedRequestID(requestID);
//             }
//             s_lastResponse = response;
//             s_lastError = err;
//             emit Response(requestID, s_lastResponse, s_lastError);
//         }
//     }

//     /**
//      * Open in Remix What is Remix? *
//      * To write a Chainlink Functions consumer contract, your contract must import FunctionsClient.sol and FunctionsRequest.sol. You can read the API references FunctionsClient and FunctionsRequest.
//      */
// }
```

These contracts are available in an NPM package, so you can import them from within your project.

```
import {FunctionsClient} from "@chainlink/contracts/src/v0.8/functions/dev/v1_0_0/FunctionsClient.sol"; import {FunctionsRequest} from
"@chainlink/contracts/src/v0.8/functions/dev/v1_0_0/libraries/FunctionsRequest.sol"; * Use the FunctionsRequest.sol library to get all the functions needed for building a Chainlink Functions request.
```

using FunctionsRequest for FunctionsRequest.Request: * The latest request id, latest received response, and latest received error (if any) are defined as state variables:

```
bytes32 public s_lastRequestId; bytes public s_lastResponse; bytes public s_lastError; * We define theResponseevent that your smart contract will emit during the callback
```

```
event Response(bytes32 indexed requestId, bytes response, bytes err); * Pass the router address for your network when you deploy the contract:
```

constructor(address router) FunctionsClient(router) * The three remaining functions are:

- `sendRequest` for sending a request. It receives the JavaScript source code, encrypted secretsUrls (in case the encrypted secrets are hosted by the user), DON hosted secrets slot id and version (in case the encrypted secrets are hosted by the DON), list of arguments to pass to the source code, subscription id, and callback gas limit as parameters. Then:
- It uses the `FunctionsRequestLibrary` to initialize the request and add any passed encrypted secrets reference or arguments. You can read the API Reference for [initializing a request](#), [adding user hosted secrets](#), [adding DON hosted secrets](#), [adding arguments](#), and [adding bytes arguments](#).

```
FunctionsRequest.Request memory req; req.initializeRequestForInlineJavaScript(source); if (encryptedSecretsUrls.length > 0) req.addSecretsReference(encryptedSecretsUrls); else if (donHostedSecretsVersion > 0) { req.addDONHostedSecrets(donHostedSecretsSlotID, donHostedSecretsVersion); } if (args.length > 0) req.setArgs(args); if (bytesArgs.length > 0) req.setBytesArgs(bytesArgs); * It sends the request to the router by calling the FunctionsClient.sendRequest function. You can read the API reference for sending a request. Finally, it stores the request id in lastRequestID then return it.
```

`s_lastRequestId = sendRequest(req.encodeCBOR(), subscriptionId, gasLimit, jobId); return s_lastRequestId;` Note: `sendRequest` accepts requests encoded in bytes. Therefore, you must encode it using `encodeCBOR` . `sendRequestCBOR` for sending a request already encoded in bytes. It receives the request object encoded in bytes, subscription id, and callback gas limit as parameters. Then, it sends the request to the router by calling the `FunctionsClient.sendRequest` function. Note: This function is helpful if you want to encode a request offchain before sending it, saving gas when submitting the request. `fulfillRequest` be invoked during the callback. This function is defined in `FunctionsClient.asVirtual(readFulfillRequest` [API reference](#)). So, your smart contract must override the function to implement the callback. The implementation of the callback is straightforward: the contract stores the latest response and error in `lastResponse` and `lastErrorResponse` after emitting the `ResponseEvent`.

```
s.lastResponse = response; s.lastError = err; emit Response(requestId, s.lastResponse, s.lastError);
```

JavaScript example

[source.js](#)

The JavaScript code is similar to the [Call Multiple Data Sources](#) tutorial.

request.is

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 seven 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, upload the encrypted secrets to the DON, and then encode the reference to the DON-hosted encrypted secrets. This is done in three 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.
22. Call the `buildDONHostedEncryptedSecretsReference` function of the `SecretsManager` instance and use the slot ID and version to encode the DON-hosted encrypted secrets reference.
23. Encode the request data offchain using the `buildRequestCBOR` function from the Functions NPM package.
24. Making a Chainlink Functions request:
25. Initialize your functions consumer contract using the contract address, abi, and ethers signer.
26. Make a [static call](#) to the `sendRequestCBOR` function of your consumer contract to return the request ID that Chainlink Functions will generate.
27. Call the `sendRequestCBOR` function of your consumer contract. Note: The encoded data that was generated by `buildRequestCBOR` is passed in the request.
28. Waiting for the response:
29. 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.
30. 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).