Encode request data offchain

This tutorial shows you how make multiple API calls from your smart contract to a Decentralized Oracle Network. AfterOCR 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 theAPI query parameters andAPI 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 theCall 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 theterms can result in suspension or termination of your Chainlink Functions account.

Prerequisites

note

You might skip these prerequisites if you have followed one of theseguides . You can check your subscription details (including the balance in LINK) in theChainlink Functions Subscription Manager . If your subscription runs out of LINK, follow theFund 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, configureNode.js , clone thesmartcontractkit/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 theLINK Token Contracts page.
4. Polygon Mumbai testnet and LINK token contract
5. Request testnet MATIC from thePolygon Faucet .
6. Request testnet LINK fromfaucets.chain.link/mumbai .

Install the required frameworks and dependencies:

1. Install the latest release of Node.js 20. Optionally, you can use thenvm package to switch between Node.js versions withnvm use 20.

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

node-v$node-vv20.9.0 2. In a terminal, clone thesmart-contract examples repository and change directories. This example repository imports theChainlink Functions Toolkit NPM package . You can import this package to your own projects to enable them to work with Chainlink Functions.

gitclone https://github.com/smartcontractkit/smart-contract-examples.git&&\cd./smart-contract-examples/functions-examples/ 3. Runnpm installto install the dependencies.

npminstall 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. Runnpx env-enc setto 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 fromAlchemy ,Infura , or another node provider service.
- PRIVATE_KEY: Find the private key for your testnet wallet. If you use MetaMask, follow the instructions toExport a Private Key .Note: Your private key is needed to sign any transactions you make such as making requests.

npx env-encset

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 onPolygon Mumbai

1. Open the FunctionsConsumerExample.sol contract in Remix.

Open in Remix What is Remix? 2. Compile the contract. 3. Open MetaMask and select thePolygon Mumbainetwork. 4. In Remix under theDeploy & Run Transactionstab, selectInjected Provider - MetaMaskin theEnvironmentlist. Remix will use the MetaMask wallet to communicate withPolygon Mumbai. 5. Under theDeploysection, fill in the router address for your specific blockchain. You can find both of these addresses on theSupported Networks page. ForPolygon Mumbai, the router address is0x6E2dc0F9DB014aE19888F539E59285D2Ea04244C. 6. Click theDeploybutton to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract toPolygon Mumbai. 7. After you confirm the transaction, the contract address appears in theDeployed Contractslist. Copy the contract address.

Create a subscription

Follow theManaging 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 atfunctions.chain.link .

Tutorial

This tutorial is configured to get the medianBTC/USDprice from multiple data sources. For a detailed explanation of the code example, read theExamine the code section.

You can locate the scripts used in this tutorial in theexamples/9-send-cbordirectory .

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 theminimum balance for uploading encrypted secrets section to learn more). You can check your subscription details (including the balance in LINK) in theChainlink Functions Subscription Manager . If your subscription runs out of LINK, follow theFund a Subscription guide. This guide recommends maintaining at least 2 LINK within your subscription.
2. Get a free API key fromCoinMarketCap and note your API key.
3. Runnpx env-enc setto add an encryptedCOINMARKETCAP_API_KEYto your.env.encfile.

npx env-encset

To run the example:

1. Open the filerequest.js, which is located in the9-send-cborfolder.
2. Replace the consumer contract address and the subscription ID with your own values.

constconsumerAddress="0x8dFf78B7EE3128D00E90611FBeD20A71397064D9"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID 3. Make a request:

nodeexamples/9-send-cbor/request.jsThe 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

Simulation result { capturedTerminalOutput: 'Median Bitcoin price: 25713.36\n', responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000000000273c48' } ✅ Decoded response to uint256: 2571336n

Estimate request costs... Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256)) Fulfillment cost estimated to 0.000000000000215 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

✅ Request 0x74b5b88db49dad155d1cd9b0da47cb140caf1f824cf09a88bc46132c21d41d20 fulfilled with code: 0. Cost is 0.000039224977086446 LINK. Complete response: { requestId: '0x74b5b88db49dad155d1cd9b0da47cb140caf1f824cf09a88bc46132c21d41d20', subscriptionId: 3, totalCostInJuels: 39224977086446n, responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000000000273c48', errorString: '', returnDataBytesHexstring: '0x', fulfillmentCode: 0 }

✅ 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 endpointhttps://01.functions-gateway.testnet.chain.link/user.
- Your request was successfully sent to Chainlink Functions. The transaction in this example is0x310c93ba1c9515a8e1dc4308e198c36f5f915410fc67c13f80ffb29a43dcbe60 and the request ID is0x74b5b88db49dad155d1cd9b0da47cb140caf1f824cf09a88bc46132c21d41d20.
- The DON successfully fulfilled your request. The total cost was:0.000039224977086446 LINK.
- The consumer contract received a response inbyteswith a value of0x0000000000000000000000000000000000000000000000000000000000273c48. Decoding it offchain touint256gives you a result:2571336. The median BTC price is 25713.36 USD.

Examine the code

FunctionsConsumerExample.sol

// SPDX-License-Identifier: MITpragmasolidity0.8.19;import{FunctionsClient}from"@chainlink/contracts/src/v0.8/functions/v1_0_0/FunctionsClient.sol";import{ConfirmedOwner}from"@chainlink/contracts/src/v0.8/shared/access/Con
*** 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.**
***/contractFunctionsConsumerExampleisFunctionsClient,ConfirmedOwner{usingFunctionsRequestforFunctionsRequest.Request;bytes32publics_lastRequestId;bytespublics_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
/functionsendRequest(stringmemorysource,bytesmemoryencryptedSecretsUrls,uint8donHostedSecretsSlotID,uint64donHostedSecretsVersion,string[]memoryargs,bytes[]memorybytesArgs,uint64subscri
{FunctionsRequest.Requestmemoryreq;req.initializeRequestForInlineJavaScript(source);if(encryptedSecretsUrls.length>0)req.addSecretsReference(encryptedSecretsUrls);elseif(donHostedSecretsVersi
{req.addDONHostedSecrets(donHostedSecretsSlotID,donHostedSecretsVersion);}if(args.length>0)req.setArgs(args);if(bytesArgs.length>0)req.setBytesArgs(bytesArgs);s_lastRequestId=_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*
/functionsendRequestCBOR(bytesmemoryrequest,uint64subscriptionId,uint32gasLimit,bytes32donID)externalonlyOwnerreturns(bytes32requestId)
{s_lastRequestId=_sendRequest(request,subscriptionId,gasLimit,donID);returns_lastRequestId;}/* ** @notice Store latest result/error * @param requestId The request ID, returned by sendRequest() * @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* /functionfulfillRequest(bytes32requestId,bytesmemoryresponse,bytesmemoryerr)internaloverride{if(s_lastRequestId!=requestId)
{revertUnexpectedRequestID(requestId);}s_lastResponse=response;s_lastError=err;emitResponse(requestId,s_lastResponse,s_lastError);}} Open in Remix What is Remix? * To write a Chainlink Functions consumer contract, your contract must importFunctionsClient.sol andFunctionsRequest.sol . You can read the API referencesFunctionsClient andFunctionsRequest .

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/v1_0_0/FunctionsClient.sol"; import {FunctionsRequest} from "@chainlink/contracts/src/v0.8/functions/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:

- sendRequestfor 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 theFunctionsRequestlibrary to initialize the request and add any passed encrypted secrets reference or arguments. You can read the API Reference forInitializing a request ,adding user hosted secrets ,adding DON hosted secrets ,adding arguments , andadding 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 theFunctionsClientsendRequestfunction. You can read the API reference forsending a request . Finally, it stores the request id ins_lastRequestIdthen return it.

s_lastRequestId = _sendRequest(req.encodeCBOR(), subscriptionId, gasLimit, jobId); return s_lastRequestId;Note:_sendRequestaccepts requests encoded inbytes. Therefore, you must encode it usingencodeCBOR . * sendRequestCBORfor sending a request already encoded inbytes. It receives the request object encoded inbytes, subscription id, and callback gas limit as parameters. Then, it sends the request to the router by calling theFunctionsClientsendRequestfunction.Note: This function is helpful if you want to encode a request offchain before sending it, saving gas when submitting the request. * fulfillRequestto be invoked during the callback. This function is defined inFunctionsClientasvirtual(readfulfillRequestAPI 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 ins_lastResponseands_lastErrorbefore emitting theResponseevent.

s_lastResponse = response; s_lastError = err; emit Response(requestId, s_lastResponse, s_lastError);

JavaScript example

source.js

The JavaScript code is similar to theCall Multiple Data Sources tutorial.

request.js

This explanation focuses on therequest.js script and shows how to use theChainlink 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 andfs : Used to read thesource 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 theNPM README .
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read theofficial documentation to learn more.

- ../abi/functionsClient.json: The abi of the contract your script will interact with.Note: The script was tested with thisFunctionsConsumerExample contract .

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

constconsumerAddress="0x8dFf78B7EE3128D00E90611FBeD20A71397064D9"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID The primary function that the script executes ismakeRequestMumbai. 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 usefs.readFileSyncto readsource.jsand then calltoString()to get the content as astringobject.
7. args: During the execution of your function, These arguments are passed to the source code. Theargsvalue 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 etherssignerandproviderobjects. 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. UsesimulateScriptfrom the Chainlink Functions NPM package.

15. Read theresponseof the simulation. If successful, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.uint256in this example).

16. Estimating the costs:

17. Initialize aSubscriptionManagerfrom the Functions NPM package, then call theestimateFunctionsRequestCostfunction.

18. The response is returned in Juels (1 LINK = 10**18 Juels). Use theethers.utils.formatEtherutility 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 aSecretsManagerinstance from the Functions NPM package, then call theencryptSecretsfunction.

21. Call theuploadEncryptedSecretsToDONfunction of theSecretsManagerinstance. This function returns an object containing asuccessboolean as long asversion, the secret version on the DON storage.
22. Call thebuildDONHostedEncryptedSecretsReferencefunction of theSecretsManagerinstance and use the slot ID and version to encode the DON-hosted encrypted secrets reference.
23. Encode the request data offchain using thebuildRequestCBORfunction 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 astatic call to thesendRequestCBORfunction of your consumer contract to return the request ID that Chainlink Functions will generate.
27. Call thesendRequestCBORfunction of your consumer contract.Note: The encoded data that was generated bybuildRequestCBORis passed in the request.

28. Waiting for the response:

29. Initialize aResponseListenerfrom the Functions NPM package and then call thelistenForResponseFromTransactionfunction to wait for a response. By default, this function waits for five minutes.

30. Upon reception of the response, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.uint256in this example).