Return multiple responses and decode them in your smart contract

In the [Using Imports with Functions](#) tutorial, we explored the fundamentals of module imports. This tutorial will teach you how to use the Ethers library [encode](#) function to perform ABI encoding of several responses. Then, you will use the [ABI specifications](#) in Solidity to decode the responses in your smart contract.

caution

Users are fully responsible for any dependencies their JavaScript source code imports. Chainlink is not responsible for any imported dependencies and provides no guarantees of the validity, availability or security of any libraries a user chooses to import or the repositories from which these dependencies are downloaded. Developers are advised to fully vet any imported dependencies or avoid dependencies altogether to avoid any risks associated with a compromised library or a compromised repository from which the dependency is downloaded.

Prerequisites

This tutorial assumes you have completed the [Using Imports with Functions](#) tutorial. Also, 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.

In this tutorial, you will use a different Chainlink Functions consumer contract, which shows how to use ABI decoding to decode the response received from Chainlink Functions:

1. [Open the FunctionsConsumerDecoder.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 this address on the [Supported 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. 8. Add your consumer contract address to your subscription onPolygon Mumbai.

Tutorial

This tutorial demonstrates using the [ethers](#) library to interact with smart contract functions through a JSON RPC provider. It involves calling the [latestRoundData](#), [decimals](#), and [description](#) functions of a price feed contract based on theAggregatorV3Interface. After retrieving the necessary data, the guide shows how to use ABI encoding to encode these responses into a single hexadecimal string and then convert this string to a [Uint8Array](#). This step ensures compliance with the [Chainlink functions' API requirements](#), which specify that the source code must return a Uint8Array representing the bytes for on-chain use.

You can locate the scripts used in this tutorial in the [examples/12-abi-encodingdirectory](#).

To run the example:

1. Make sure you have correctly set up your environment first. If you haven't already, follow the [Set up your environment](#) section of the [Using Imports with Functions](#) tutorial.
2. Open the filerequest.js, located in the [12-abi-encoding](#) folder.
3. Replace the consumer contract address and the subscription ID with your own values.

constconsumerAddress="0x5fC6e53646CC53f0C3575fd2c71b5056c4823f5c"// REPLACE this with your Functions consumer addressconstsubscriptionId=139// REPLACE this with your subscription ID 4. Make a request:

nodeexamples/12-abi-encoding/request.jsThe script runs your function in a sandbox environment before making an onchain transaction:

$ node examples/12-abi-encoding/request.js secp256k1 unavailable, reverting to browser version Start simulation... Simulation result { capturedTerminalOutput: 'Fetched BTC / USD price: dataFeedResponse.answer\n' + 'Updated at: 1707155795\n' + 'Decimals: 8\n' + 'Description: BTC / USD\n', responseBytesHexstring: '0x000000000000000000000000000000000000000000000000000003e1b870b6320000000000000000000000000000000000000000000000000000000065c121530000000000000000000000000000000000
} ✅ Decoded response to bytes: 0x000000000000000000000000000000000000000000000000000003e1b870b6320000000000000000000000000000000000000000000000000000000065c121530000000000000000000000000000000000

Estimate request costs... Fulfillment cost estimated to 0.20180840394238 LINK

Make request...

✅ Functions request sent! Transaction hash 0x660bc9bd4c85645209295c957d0764823b94a1d351a5ead380113e5352e01fb5. Waiting for a response... See your request in the explorer https://mumbai.polygonscan.com/tx/0x660bc9bd4c85645209295c957d0764823b94a1d351a5ead380113e5352e01fb5

✅ Request 0xa790aaaca3fdcdd0ca0b0e68bf0c699d84da8d59f8b6dd95aa3d4046a3801c27 successfully fulfilled. Cost is 0.200304341399547731 LINK.Complete reponse: { requestId: '0xa790aaaca3fdcdd0ca0b0e68bf0c699d84da8d59f8b6dd95aa3d4046a3801c27', subscriptionId: 139, totalCostInJuels: 200304341399547731n, responseBytesHexstring: '0x000000000000000000000000000000000000000000000000000003e1b870b6320000000000000000000000000000000000000000000000000000000065c121530000000000000000000000000000000000
errorString: '', returnDataBytesHexstring: '0x', fulfillmentCode: 0 }

✅ Raw response: 0x000000000000000000000000000000000000000000000000000003e1b870b6320000000000000000000000000000000000000000000000000000000065c121530000000000000000000000000000000000

✅ Fetched BTC / USD price: 4267996919346 (updatedAt: 1707155795) (decimals: 8) (description: BTC / USD)The 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.
- Your request was successfully sent to Chainlink Functions. The transaction in this example is0x660bc9bd4c85645209295c957d0764823b94a1d351a5ead380113e5352e01fb5, and the request ID is0xa790aaaca3fdcdd0ca0b0e68bf0c699d84da8d59f8b6dd95aa3d4046a3801c27.
- The DON successfully fulfilled your request. The total cost was:0.200304341399547731 LINK.
- The consumer contract received a response in hexadecimal string with a value of0x000000000000000000000000000000000000000000000000000003e1b870b6320000000000000000000000000000000000000000000000000000000065c12153000000000000000000000000000 This value is the ABI encoded response of thelatestRoundData,decimals, anddescriptionof theBTC / USDprice feed. This value is then decoded and stored in the consumer contract.
- The script calls the consumer contract to fetch the decoded values and then logs them to the console. The output isFetched BTC / USD price: 4267996919346 (updatedAt: 1707155795) (decimals: 8) (description: BTC / USD).

Examine the code

FunctionsConsumerDecoder.sol

// SPDX-License-Identifier: MITpragmasolidity0.8.19;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/
*** 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.**
***/contractFunctionsConsumerDecoderisFunctionsClient,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;}/* *@dev Internal function to process the outcome of a data request. It stores the latest response or error and updates the contract state accordingly. This function is designed to handle only one of `response` or `err` at a time, not both. It decodes the response if present and emits events to log both raw and decoded data. * * @param requestId The unique identifier of the request, originally returned by `sendRequest`. Used to match responses with requests. * @param response The raw aggregated response data from the external source. This data is ABI-encoded and is expected to contain specific information (e.g., answer, updatedAt) if no error occurred. The function attempts to decode this data if `response` is not empty. * @param err The raw aggregated error information, indicating an issue either from the user's code or within the execution of the user Chainlink Function. * * Emits a `DecodedResponse` event if the `response` is successfully decoded, providing detailed information about the data received. * Emits a `Response` event for every call to log the raw response and error data. * * Requirements: * - The `requestId` must match the last stored request ID to ensure the response corresponds to the latest request sent. * - Only one of `response` or `err` should contain data for a given call; the other should be empty.* /functionfulfillRequest(bytes32requestId,bytesmemoryresponse,bytesmemoryerr)internaloverride{if(s_lastRequestId!=requestId)
{revertUnexpectedRequestID(requestId);}s_lastError=err;s_lastResponse=response;if(response.length>0)
{(uint256answer,uint256updatedAt,uint8decimals,stringmemorydescription)=abi.decode(response,

(uint256,uint256,uint8,string));s_answer=answer;s_updatedAt=updatedAt;s_decimals=decimals;s_description=description;emitDecodedResponse(requestId,answer,updatedAt,decimals,description);}emi Open in Remix What is Remix? This Solidity contract is similar to theFunctionsConsumer.sol contract used in theUsing Imports with Functions tutorial. The main difference is the processing of the response in thefulfillRequestfunction:

- It uses Solidityabi.decodeto decode theresponseto retrieve theanswer,updatedAt,decimals, anddescription.

(uint256answer,uint256updatedAt,uint8decimals,stringmemorydescription)=abi.decode(response,(uint256,uint256,uint8,string)); * Then stores the decoded values in the contract state.

s_answer=answer;s_updatedAt=updatedAt;s_decimals=decimals;s_description=description;

JavaScript example

source.js

The Decentralized Oracle Network will run theJavaScript code . The code is self-explanatory and has comments to help you understand all the steps.

note

Functions requests with custom source code can use vanillaDeno . Import statements and imported modules are supported only on testnets. You cannot use any require statements.

It is important to understand that importing an NPM package into Deno does not automatically ensure full compatibility. Deno and Node.js have distinct architectures and module systems. While some NPM packages might function without issues, others may need modifications or overrides, especially those relying on Node.js-specific APIs or features Deno does not support.

The examplesource.jsfile is similar to the one used in theUsing Imports with Functions tutorial. It uses a JSON RPC call to thelatestRoundData ,decimals , anddescription functions of aChainlink Data Feed . It then uses theetherslibrary to encode the response of these functions into a single hexadecimal string.

constencoded=ethers.AbiCoder.defaultAbiCoder().encode(["uint256","uint256","uint8","string"],[dataFeedResponse.answer,dataFeedResponse.updatedAt,decimals,description]) Finally, it uses theetherslibrarygetBytes to convert the hexadecimal string to aUint8Array:

returnethers.getBytes(encoded)

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/functionsDecoder.json: The abi of the contract your script will interact with.Note: The script was tested with thisFunctionsConsumerDecoder contract .

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

constconsumerAddress="0x5fC6e53646CC53f0C3575fd2c71b5056c4823f5c"// REPLACE this with your Functions consumer addressconstsubscriptionId=139// REPLACE this with your subscription ID The primary function that the script executes ismakeRequestMumbai. This function consists of five 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. explorerUrl: Block explorer url of Polygon Mumbai.
5. 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.
6. args: During the execution of your function, These arguments are passed to the source code.
7. gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
8. Initialization of etherssignerandproviderobjects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.
9. Simulating your request in a local sandbox environment:
10. UsesimulateScriptfrom the Chainlink Functions NPM package.
11. Read theresponseof the simulation. If successful, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.bytesin this example).
12. Estimating the costs:
13. Initialize aSubscriptionManagerfrom the Functions NPM package, then call theestimateFunctionsRequestCost.
14. The response is returned in Juels (1 LINK = 10**18 Juels). Use theethers.utils.formatEtherutility function to convert the output to LINK.
15. Making a Chainlink Functions request:
16. Initialize your functions consumer contract using the contract address, abi, and ethers signer.
17. Call thesendRequestfunction of your consumer contract.
18. Waiting for the response:
19. Initialize aResponseListenerfrom the Functions NPM package and then call thelistenForResponseFromTransactionfunction to wait for a response. By default, this function waits for five minutes.
20. Upon reception of the response, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.bytesin this example).
21. Read the decoded response:
22. Call thes_answer,s_updatedAt,s_decimals, ands_descriptionfunctions of your consumer contract to fetch the decoded values.
23. Log the decoded values to the console.

Handling complex data types with ABI Encoding and Decoding

This section details the process of encoding complex data types intoUint8Arraytyped arrays to fulfill the Ethereum Virtual Machine (EVM) data handling requirements for transactions and smart contract interactions. It will then outline the steps for decoding these byte arrays to align with corresponding structures defined in Solidity.

Consider a scenario where a contract needs to interact with a data structure that encapsulates multiple properties, including nested objects:

{"id":1,"metadata":{"description":"Decentralized Oracle Network","awesome":true}} Transferring and storing this kind of structured data requires encoding it into a format (array of 8-bit unsigned integers) that smart contracts can accept and process.

Encoding in JavaScript

Because Chainlink Functions supports important external modules, you can import a web3 library such asethers.jsand perform encoding. To encode complex data structures, you can use thedefaultAbiCoder.encode function from theethers.jslibrary. The function takes two arguments:

- An array of Solidity data types.
- The corresponding data in JavaScript format.

and returns the encoded data as a hexadecimal string.

Here's how you can encode the aforementioned complex data:

const{ethers}=awaitimport("npm: ")// Import ethers.js v6.10.0constabiCoder=ethers.AbiCoder.defaultAbiCoder()// Define the data structureconstcomplexData={id:1,metadata:{description:"Decentralized Oracle Network",awesome:true,},}// Define the Solidity types for encodingconsttypes=["tuple(uint256 id, tuple(string description, bool awesome) metadata)"]// Encoding the dataconstencodedData=abiCoder.encode(types,[complexData]) After encoding the data, it's necessary to format it as aUint8Arrayarray for smart contract interactions and blockchain transactions. In Solidity, the data type for byte arrays data isbytes. However, when working in a JavaScript environment, such as when using theethers.jslibrary, the equivalent data structure is aUint8Array.

Theethers.jslibrary provides thegetBytes function to convert encoded hexadecimal strings into aUint8Array:

returnethers.getBytes(encodedData)// Return the encoded data converted into a Uint8Array

Decoding in Solidity

The encoded data can be decoded using theabi.decodefunction. To decode the data, you'll need to handle the decoding in yourfulfillRequestfunction:

// SPDX-License-Identifier: MITpragmasolidity0.8.19;contractDataDecoder{// Example of a structure to hold the complex datastructMetadata{stringdescription;boolawesome;}structComplexData{uint256id;Metadata metadata;}// ... other contract functions (including the send request function)// Fulfill function (callback function)functionfulfillRequest(bytes32requestId,bytesmemoryresponse,bytesmemoryerr)internaloverride{// Decode the responseComplexDatamemorymetadata=abi.decode(response,(ComplexData));// ... rest of the function}}