Using Imports with Functions

This tutorial demonstrates how to import modules and use them with your Functions source code. Modules that are imported into Functions source code must meet the following requirements:

- Each import must be 10 MB or less in size.
- Up to 100 imports total are supported.
- Deno supportsESM compatible NPM imports and somestandard Node modules . See theCompatability List for details.

- Third-party modules are imported at runtime, so import statements must use asynchronous logic like the following examples:

- Importing fromdeno.land:

const{escape}=awaitimport("https://deno.land/std/regexp/mod.ts") * ESM-compatible packages:

const{format}=awaitimport("npm:date-fns") * Standard Node modules:

constpath=awaitimport("node:path") * CDN imports:

constlodash=awaitimport("http://cdn.skypack.dev/lodash") * Imported modules abide by all sandbox restrictions and do not have access to the file system, environment variables, or any other Deno permissions.

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

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 example importsethers and demonstrates how to call a smart contract functions using a JSON RPC provider to call an onchain function. In this example, the source code calls thelatestRoundData()function from theAggregatorV3Interface. Read theExamine the code section for a detailed description of the code example.

You can locate the scripts used in this tutorial in theexamples/11-package-importsdirectory .

To run the example:

1. Open the filerequest.js, which is located in the11-package-importsfolder.
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/11-package-imports/request.jsThe script runs your function in a sandbox environment before making an onchain transaction:

$ node examples/11-package-imports/request.js secp256k1 unavailable, reverting to browser version Start simulation... Simulation result { capturedTerminalOutput: 'Fetched BTC / USD price: 4367193987453\n', responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000000003f8d10bd97d' } ✅ Decoded response to int256: 4367193987453n

Estimate request costs... Fulfillment cost estimated to 0.20243353895715 LINK

Make request...

✅ Functions request sent! Transaction hash 0xed3d0419189c012ce852b37b51d47bdcd80f06a4749b4c01a81a3f5fb06139e3. Waiting for a response... See your request in the explorer https://mumbai.polygonscan.com/tx/0xed3d0419189c012ce852b37b51d47bdcd80f06a4749b4c01a81a3f5fb06139e3

✅ Request 0xa8a7bec42edc16cf549f69e734161f2f2550a1057bb4060611a8043253ee61ef successfully fulfilled. Cost is 0.200014890551438381 LINK.Complete reponse: { requestId: '0xa8a7bec42edc16cf549f69e734161f2f2550a1057bb4060611a8043253ee61ef', subscriptionId: 38, totalCostInJuels: 200014890551438381n, responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000000003f8d10bd97d', errorString: '', returnDataBytesHexstring: '0x', fulfillmentCode: 0 }

✅ Decoded response to int256: 4367193987453nThe 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 is0xed3d0419189c012ce852b37b51d47bdcd80f06a4749b4c01a81a3f5fb06139e3and the request ID is0xdfb161de5a6ad34e58bb115dd07651a11d4cf4739652f509ecad78a1bf506e82.
- The DON successfully fulfilled your request. The total cost was:0.200435783655508510 LINK.
- The consumer contract received a response inbyteswith a value of0x0000000000000000000000000000000000000000000000000000000003f8d10bd97d. Decoding it offchain toint256gives you a result:4367193987453.

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 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 uses a JSON RPC call to thelatestRoundData()function of aChainlink Data Feed .

The request requires a few modifications to work in the Chainlink Functions environment. For example, theJsonRpcProviderclass must be inherited to override the JsonRpcProvidersendmethod . This customization is necessary because Deno does not natively support Node.js modules likehttp orhttps . We override the_sendmethod to use thefetch API , which is the standard way to make HTTP(s) requests in Deno.Note: Theurlpassed in the constructor is the URL of the JSON RPC provider.

// Chainlink Functions compatible Ethers JSON RPC provider class// (this is required for making Ethers RPC calls with Chainlink Functions)classFunctionsJsonRpcProviderextendsethers.JsonRpcProvider{constructor(url){super(url)this.url=url}async_send(payload){letresp=awaitfetch(this.url,{method:"POST",headers:{"Content-Type":"application/json"},body:JSON.stringify(payload),})returnresp.json()}} After the class is extended, you can initialize the provider object with theRPC_URLand await the response.

constprovider=newFunctionsJsonRpcProvider(RPC_URL)constdataFeedContract=newethers.Contract(CONTRACT_ADDRESS,abi,provider)constdataFeedResponse=awaitdataFeedContract.latestRour In this example, the contract returns anint256value. Encode the value so request.js can properly decode it.

returnFunctions.encodeInt256(dataFeedResponse.answer)

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="0x91257aa1c6b7f382759c357fbc53c565c80f7fee"// REPLACE this with your Functions consumer addressconstsubscriptionId=38// 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.int256in 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.int256in this example).