

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 supports [ESM compatible NPM imports](#) and some [standard Node modules](#). See the [Compatability List](#) for details.
- Third-party modules are imported at runtime, so import statements must use asynchronous logic like the following examples:
- Importing from [deno.land](#):

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

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

```
const path=awaitimport("node:path") * CDN imports:
```

`const lodash=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 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.

`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 example imports [ethers](#) 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 the [latestRoundData\(\)](#) function from the `AggregatorV3Interface`. Read the [Examine the code](#) section for a detailed description of the code example.

You can locate the scripts used in this tutorial in the [examples/11-package-imports](#) directory.

To run the example:

1. Open the `filerequest.js`, which is located in the `11-package-imports` folder.
2. Replace the consumer contract address and the subscription ID with your own values.

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

`node examples/11-package-imports/request.js` The script runs your function in a sandbox environment before making an onchain transaction:

[illegible]

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>

[illegible]

✓ Decoded response to int256: 4367193987453nThe output of the example gives you the following information:

- [illegible]

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 FunctionsConsumerExampleIsFunctionsClient, ConfirmedOwner {
    using FunctionsRequest for FunctionsRequest;
    bytes32 public s_lastRequestId;
    bytes public s_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
     */
    function sendRequest(
        string memory source,
        bytes memory encryptedSecretsUrls,
        uint8 donHostedSecretsSlotID,
        uint64 donHostedSecretsVersion,
        string[] memory args,
        bytes[] memory bytesArgs,
        uint64 subscriptionId
    ) public {
        FunctionsRequest req = FunctionsRequest({
            req.initializeRequestForInlineJavaScript(source),
            if(encryptedSecretsUrls.length > 0) req.addSecretReference(encryptedSecretsUrls),
            if(donHostedSecretsVersion > 0) 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
                 */
                req,
                subscriptionId,
                gasLimit,
                donID
            )
        });

        s_lastRequestId = req.getResponse();
        s_lastResponse = req.getResponse();

        /*
         * @notice Store latest result/error
         * @param requestId The request ID, returned by sendRequest()
         */
        s_lastRequestId = requestId;
        s_lastResponse = response;
        s_lastError = error;

        emit Response(requestId, s_lastResponse, s_lastError);

        /*
         * @notice Open in Remix What is Remix?
         * To write a Chainlink Functions consumer contract, your contract must import FunctionsClient and FunctionsRequest.
         * 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 `FunctionsRequest` library 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 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: 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.sendRequestFunction. 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 in FunctionsClient.asVirtual(readonly fulfillRequest 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 lastErrorMessage emitting the ResponseEvent.`

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

JavaScript example

[source.js](#)

The Decentralized Oracle Network will run the [JavaScript 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 vanilla [Deno](#) . 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.isfile` uses a JSON RPC call to the `latestRoundData()` function of a [Chainlink Data Feed](#).

The request requires a few modifications to work in the Chainlink Functions environment. For example, the `JsonRpcProvider` class must be inherited to override the `JsonRpcProvider.sendMethod`. This customization is necessary because Deno does not natively support Node.js modules like <https://node.dev>. We override the `_sendMethod` to use the [fetch API](#), which is the standard way to make HTTP(s) requests in Deno. Note: The URL passed 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.
```

```
const provider = new Functions.JsonRpcProvider(RPC_URL)
const dataFeedContract = new ethers.Contract(CONTRACT_ADDRESS, abi, provider)
const dataFeedResponse = await dataFeedContract.latestRoundData()

// In this example, the contract returns an int256 value. Encode the value so request.js can properly decode it.
```

```
returnFunctions.encodeInt256(dataFeedResponse.answer)
```

[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="0x91257aa1c6b7f382759c357fbc53c565c80f7fee"` // REPLACE this with your Functions consumer address
`const subscriptionId=38` // REPLACE this with your subscription ID
The primary function that the script executes is `makeRequestMumbai`. 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 use `fs.readFileSync` to read `source.js` and then call `toString()` to get the content as a string object.
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 `ethers signer` and `provider` objects. 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. Use `simulateScript` from the Chainlink Functions NPM package.
11. 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.int256` in this example).
12. Estimating the costs:
13. Initialize a `SubscriptionManager` from the Functions NPM package, then call the `estimateFunctionsRequestCost`.
14. The response is returned in `Juels` (1 LINK = 10^{18} Juels). Use the `ethers.utils.formatEther` utility 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 the `sendRequest` function of your consumer contract.
18. Waiting for the response:
19. 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.
20. 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.int256` in this example).