

POST Data to an API

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the [Countries information GraphQL API](#). After [OCR](#) completes offchain computation and aggregation, it returns the name, capital, and currency for the specified country to your smart contract. Because the endpoint is a GraphQL API, write a function that sends a GraphQL query in a [POST HTTP method](#).

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.

Maximum response size

You can return any number of responses as long as they are encoded in `abysresponse`. The maximum response size that you can return is 256 bytes.

Side effects

Building non-idempotent requests, such as sending an email or storing data on the cloud, is currently not recommended. An HTTP method is idempotent if the intended effect on the server when you make a single request is the same as the effect when you make several identical requests. Each oracle node runs the same computation in the [Offchain Reporting protocol](#). If your Chainlink Function makes non-idempotent requests, it will cause redundant requests such as sending multiple emails or storing the same data multiple times.

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 `0x6E2dc0F9DB014aE1988F539E59285D2Ea04244C`. 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 country name, capital, and currency from [countries.trevorblades.com](#) in one request. 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/4-post-data](#) directory.

To run the example:

1. Open the `filerequest.js`, which is located in the `4-post-data` folder.
2. Replace the consumer contract address and the subscription ID with your own values.

`const consumerAddress = "0x8dF78B7EE3128D00E90611FBED20A71397064D9" // REPLACE this with your Functions consumer address`
`const subscriptionId = 3 // REPLACE this with your subscription ID` 3. Make a request:

`node examples/4-post-data/request.js` The script runs your function in a sandbox environment before making an onchain transaction:

`$ node examples/4-post-data/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: 'Get name, capital and currency for country code: JP\n' + 'HTTP POST Request to https://countries.trevorblades.com/\n' + 'country response { country: { name: "Japan", capital: "Tokyo", currency: "JPY" } }\n', responseBytesHexstring: '0x7b226e616d65223a224a6170616e222c226361706974616c223a2254616b796f222c2263797272656e6379223a224a5059227d' } ✓ Decoded response to string:

The main steps of the scripts are:

- Fetch the country code from args.
- Construct the HTTP object countryRequest using `Functions.makeHttpRequest`.
- Run the HTTP request.
- Read the country name, capital, and currency from the response.
- Construct a JSON object:

`const result = {name: countryData.country.name, capital: countryData.country.capital, currency: countryData.country.currency,} * Convert the JSON object to a JSON string using JSON.stringify(result). This step is mandatory before encoding string to bytes. * Return the result as a buffer using the Functions.stringHelper function. Note: Read this article if you are new to Javascript Buffers and want to understand why they are important.`

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 [Functions Consumer Example contract](#) .

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

`const consumerAddress = "0x8dF78B7EE3128D00E90611FBED20A71397064D9"` // 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 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. The `args` value is `["JP"]`, which fetches country data for Japan.
7. `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
8. Initialization of `ethersigner` 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.string` 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.string` in this example).