Back

Automate your Functions (Time-based Automation)

This tutorial shows you how to use Chainlink Automation to automate your Chainlink Functions. Automation is essential when you want to trigger the same function regularly, such as fetching weather data daily or fetching an asset price on every block.

Read the API multiple calls tutorial before you follow the steps in this example. This tutorial uses the same example but with an important difference:

- You will deploy AutomatedFunctionsConsumerExample.sol instead of the FunctionsConsumerExample contract.

  After you deploy and set up your contract, Chainlink Automation triggers your function according to a time schedule.

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 the terms can result in suspension or termination of your Chainlink Functions account.

Prerequisites

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.enc` file 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.
   - Polygon Mumbai testnet and LINK token contract
4. Request testnet MATIC from the Polygon Faucet.
5. 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 -v
v20.9.0
```

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.enc` file 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?

4. In Remix under the **Deploy & Run Transactions** 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 tutorial is configured to get the median `BTC/USD` price from multiple data sources according to a time schedule. 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/10-automate-functions* directory.

1. Make sure to understand the API multiple calls guide.
2. 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 the minimum balance for uploading encrypted secrets section to learn more). 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

3. Get a free API key from CoinMarketCap and note your API key.
4. Run `npx env-enc set` to add an encrypted `COINMARKETCAP_API_KEY` to your `.env.enc` file.

```
npx env-enc set
```

Deploy an Automated Functions Consumer contract

1. Deploy a Functions consumer contract on *Polygon Mumbai*:
   1. Open the AutomatedFunctionsConsumerExample.sol in Remix.

   Open in Remix | What is Remix?

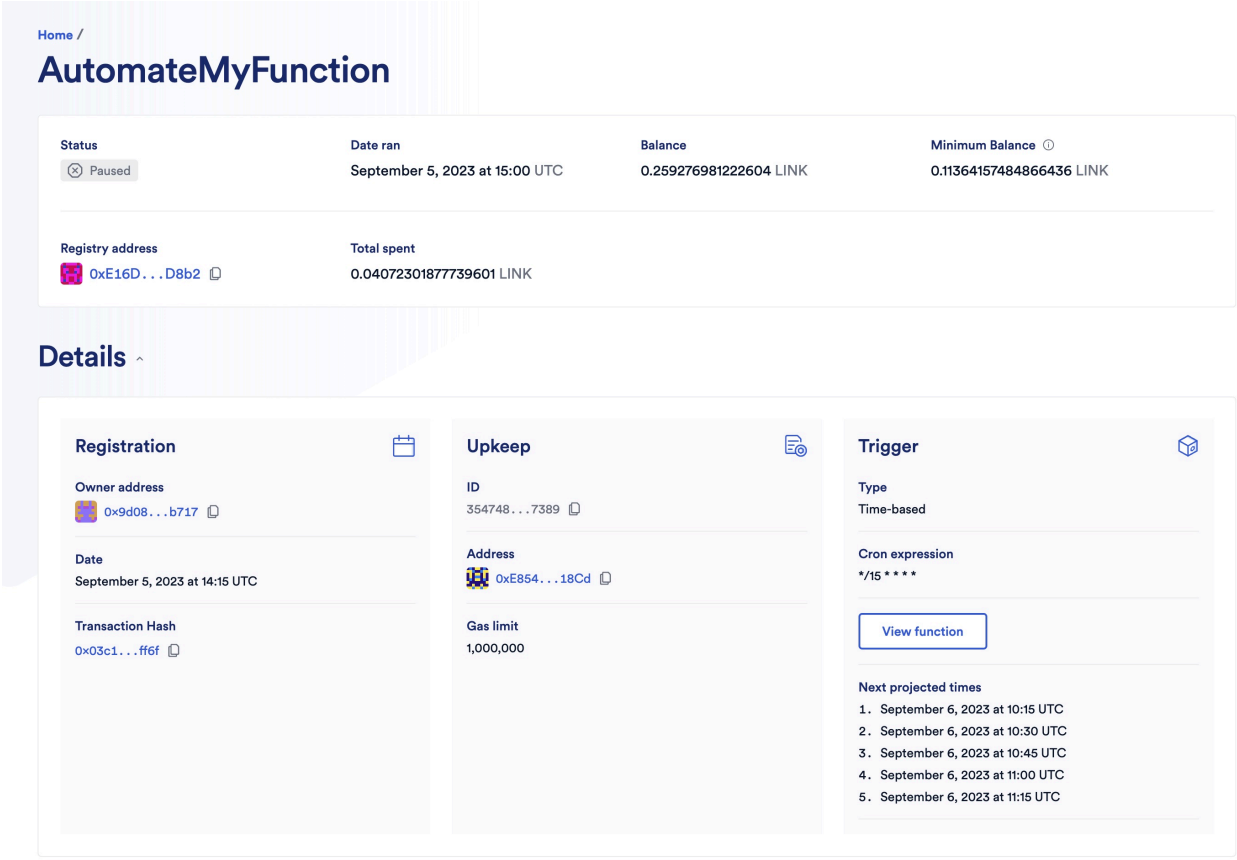
   1. Compile the contract.
   2. Open MetaMask and select the *Polygon Mumbai* network.
   3. In Remix under the **Deploy & Run Transactions** tab, select *Injected Provider - MetaMask* in the **Environment** list. Remix will use the MetaMask wallet to communicate with *Polygon Mumbai*.
   4. Under the **Deploy** section, fill in the router address for your specific blockchain. You can find this address on the Supported Networks page. For *Polygon Mumbai*, the router address is `0x6E2dc0F9DB014aE19888F539E59285D2Ea04244C`.
   5. 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*.
   6. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy your contract address.
2. Create a Chainlink Functions subscription and add your contract as an approved consumer contract. **Note**: If you followed the previous tutorials, then you can reuse your existing subscription.

Configure Chainlink Automation

deployed contract using the Chainlink Automation App. Use the following upkeep settings:

- Trigger: Time-based
- Target contract address: The address of the Chainlink Functions consumer contract that you deployed
- ABI: copy/paste the abi from automatedFunctions.json
- Target function: sendRequestCBOR
- Time interval: Every 15 minutes
- Gas limit: `1000000`
- Starting balance (LINK): `1`

You can leave the other settings at their default values for the example in this tutorial. **Note**: After creation, check your upkeep details and note the address of the upkeep contract. The upkeep contract is responsible for calling your Functions consumer contract at regular times intervals.



MONITOR YOUR BALANCES

your Chainlink Functions means they will be regularly triggered, so monitor and fund your subscription account regularly. You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager.

- Your upkeep balance: You can check this balance on the Chainlink Automation App. The upkeep balance pays Chainlink Automation Network to send your requests according to your provided time interval. Chainlink Automation will not trigger your requests if your upkeep balance runs low.

Configure your Automation Consumer contract

Two important steps are done here:

1. Configure your contract so only the upkeep contract can call the `sendRequestCBOR` function. This security measure is important to prevent anyone from calling several times `sendRequestCBOR` and draining your Functions subscription balance. Follow these steps:
   1. On RemixIDE, under the *Deploy & Transactions* tab, locate your deployed Functions consumer contract.
   2. Open the list of functions.
   3. Fill in the `setAutomationCronContract` function with the upkeep contract address you copied from the previous step.
   4. Click on *transact*. A Metamask popup appears and asks you to confirm the transaction.
   5. Confirm the transaction and wait for it to be confirmed.
2. Configure the request details by calling the `updateRequest` function. This step stores the encoded request (source code, reference to encrypted secrets if any, arguments), gas limit, subscription ID, and job ID in the contract storage (see Examine the code). To do so, follow these steps:
   1. On a terminal, go to the *Functions tutorials* directory.
   2. Open updateRequest.js and replace the consumer contract address and the subscription ID with your own values:

```
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" // REPLACE wi
const subscriptionId = 3 // REPLACE this with your subscription ID
```

EN

Example:

```
$ node examples/10-automate-functions/updateRequest.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: 25702.07\n',
  responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000(
}
✅ Decoded response to uint256:  2570207n

Make request...
Upload encrypted secret to gateways https://01.functions-gateway.testnet.chain.

✅ Secrets uploaded properly to gateways https://01.functions-gateway.testnet.c

✅ Automated Functions request settings updated! Transaction hash 0x6017e4b36f8
```

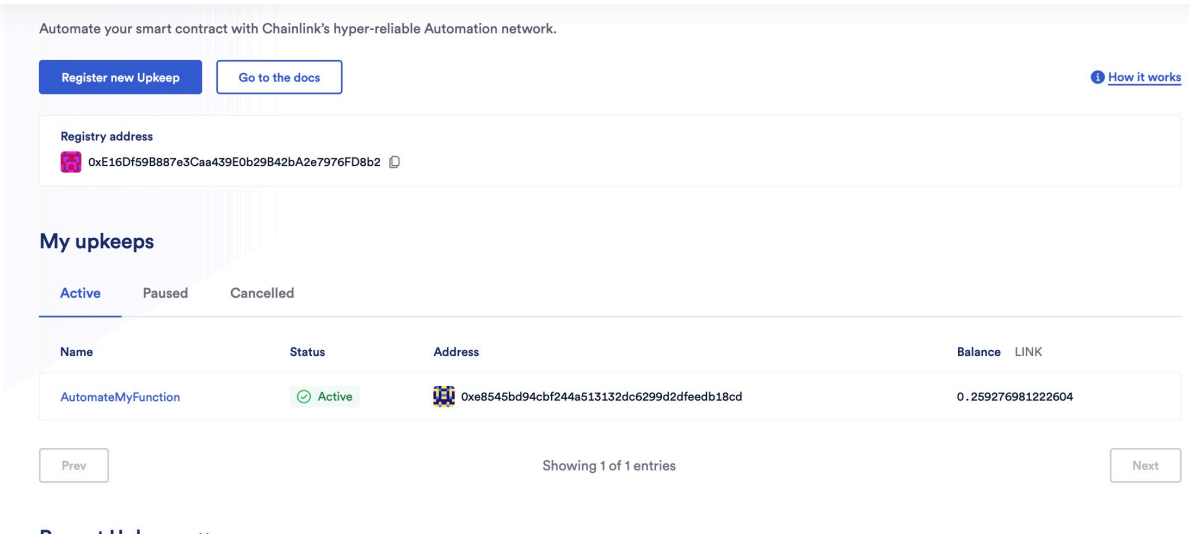
The output of the example gives you the following information:

- Your request is first run on a sandbox environment to ensure it is correctly configured.
- The encrypted secrets were uploaded to the secrets endpoint `https://01.functions-gateway.testnet.chain.link/user`.
- The Functions consumer contract's request details are updated.

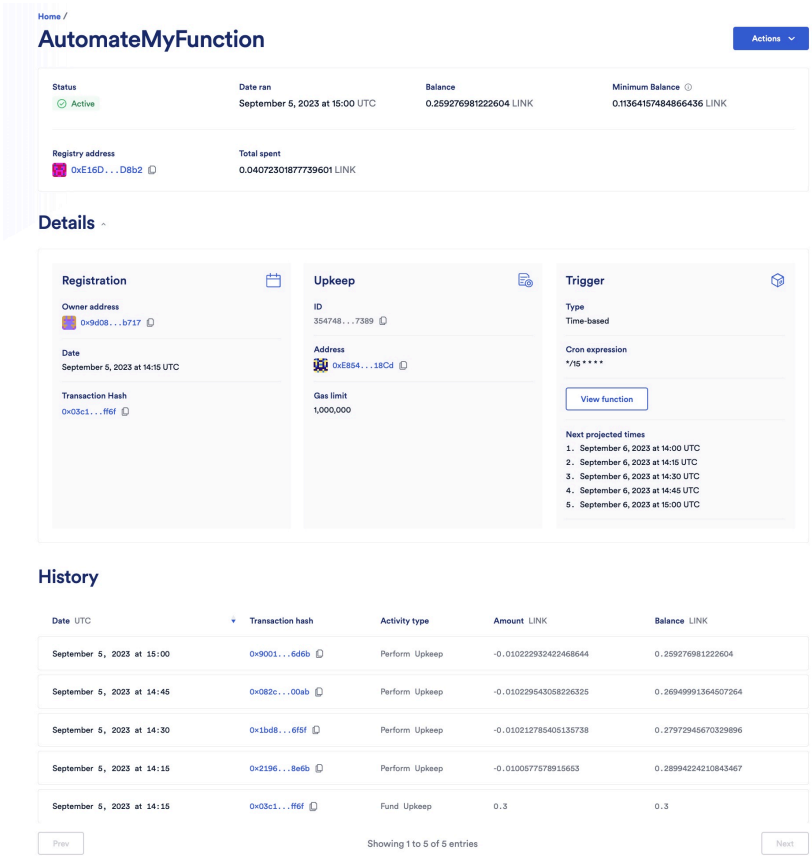
At this stage, your Functions consumer contract is configured to get the median Bitcoin price every 15 minutes.

Check Result

Go to the Chainlink Automation App and connect to Polygon Mumbai. Your upkeep will be listed under *My upkeeps*:



Click on your upkeep to fetch de details:



As you can see in the *History* table, the upkeep is running every 15 minutes. On your terminal, run the readLatest to read the latest received response:

1. Open `readLatest.js` and replace the consumer contract address with your own values:

```
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" // REPLACE this v yo
```

EN

Example:

```
$ node examples/10-automate-functions/readLatest.js
secp256k1 unavailable, reverting to browser version
Last request ID is 0x38687b48c9d8dfc7115d516840d45089a5409557765f031c14b332913e92ee7e
✅ Decoded response to uint256:  2570207n
```

Clean up

After you finish the guide:

1. Cancel your upkeep from the Chainlink Automation App. **Note**: Remember to withdraw funds after you cancel the upkeep. There is a 50-block delay once upkeep between the moment you cancel your upkeep and the moment you can withdraw funds.

Examine the code

AutomatedFunctionsConsumer.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import {FunctionsClient} from "@chainlink/contracts/src/v0.8/functions/v1_0_0/FunctionsClient.sol";
import {ConfirmedOwner} from "@chainlink/contracts/src/v0.8/shared/access/ConfirmedOwner.sol";

/**
 * @title Functions contract used for Automation.
 * @notice This contract is a demonstration of using Functions and Automation.
 * @notice You may need to add a Forwarder for additional security.
 * @notice NOT FOR PRODUCTION USE
 */
contract AutomatedFunctionsConsumerExample is FunctionsClient, ConfirmedOwner {
    address public upkeepContract;
    bytes public request;
    uint64 public subscriptionId;
    uint32 public gasLimit;
    bytes32 public donID;
    bytes32 public s_lastRequestId;
    bytes public s_lastResponse;
```

```
        address caller,
        address owner,
        address automationRegistry
    );
    error UnexpectedRequestID(bytes32 requestId);

    event Response(bytes32 indexed requestId, bytes response, bytes err);

    constructor(
        address router
    ) FunctionsClient(router) ConfirmedOwner(msg.sender) {}

    /**
* @notice Reverts if called by anyone other than the contract owner or automation registry.
*/
    modifier onlyAllowed() {
        if (msg.sender != owner() && msg.sender != upkeepContract)
            revert NotAllowedCaller(msg.sender, owner(), upkeepContract);
        _;
    }

    function setAutomationCronContract(
        address _upkeepContract
    ) external onlyOwner {
        upkeepContract = _upkeepContract;
    }

    /// @notice Update the request settings
    /// @dev Only callable by the owner of the contract
    /// @param _request The new encoded CBOR request to be set. The request is encoded offchain
    /// @param _subscriptionId The new subscription ID to be set
    /// @param _gasLimit The new gas limit to be set
    /// @param _donID The new job ID to be set
    function updateRequest(
        bytes memory _request,
        uint64 _subscriptionId,
        uint32 _gasLimit,
        bytes32 _donID
    ) external onlyOwner {
        request = _request;
        subscriptionId = _subscriptionId;
        gasLimit = _gasLimit;
        donID = _donID;
    }

    /**
* @notice Send a pre-encoded CBOR request
* @return requestId The ID of the sent request
```

EN

```
        onlyAllowed
        returns (bytes32 requestId)
    {
        s_lastRequestId = _sendRequest(
            request,
            subscriptionId,
            gasLimit,
            donID
        );
        return s_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
    */
    function fulfillRequest(
        bytes32 requestId,
        bytes memory response,
        bytes memory err
    ) internal override {
        if (s_lastRequestId != requestId) {
            revert UnexpectedRequestID(requestId);
        }
        s_lastResponse = response;
        s_lastError = err;
        emit Response(requestId, s_lastResponse, s_lastError);
    }
}
```

Open in Remix | What is Remix?

- To write an automated Chainlink Functions consumer contract, your contract must import FunctionsClient.sol. You can read the API reference of FunctionsClient.

  The contract is available in an NPM package, so you can import it from within your project.

  ```
  import {FunctionsClient} from "@chainlink/contracts/src/v0.8/functions/v1_0_0/Fu
  ```

- The `upkeepContract` address is stored in the contract storage. The contract owner sets this variable by calling the `setAutomationCronContract` function. **Note**: This variable is

```
address public upkeepContract
```

- The encoded `request`, `subscriptionId`, `gasLimit`, and `jobId` are stored in the contract storage. The contract owner sets these variables by calling the `updateRequest` function. **Note**: The request (source code, secrets, if any, and arguments) is encoded offchain.
- 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 the `Response` event 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 two remaining functions are:
  - `sendRequestCBOR` for sending a request already encoded in `bytes`. It sends the request to the router by calling the `FunctionsClient sendRequest` function.
  - `fulfillRequest` to be invoked during the callback. This function is defined in `FunctionsClient` as `virtual` (read `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 `s_lastResponse` and `s_lastError` before emitting the `Response` event.

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

source.js

The JavaScript code is similar to the Call Multiple Data Sources tutorial.

This explanation focuses on the update.js script and shows how to use the Chainlink Functions NPM package in your own JavaScript/TypeScript project to encode a request offchain then store in your contract. 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/automatedFunctions.json` : The abi of the contract your script will interact with.

  Note: The script was tested with this AutomatedFunctionsConsumer contract.

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

```
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" // REPLACE this v yo
const subscriptionId = 3 // REPLACE this with your subscription ID
```

The primary function that the script executes is `updateRequestMumbai` . This function consists of five main parts:

1. Definition of necessary identifiers:
   - `routerAddress` : Chainlink Functions router address on Polygon Mumbai.
   - `donId` : Identifier of the DON that will fulfill your requests on Polygon Mumbai.
   - `gatewayUrls` : The secrets endpoint URL to which you will upload the encrypted secrets.
   - `explorerUrl` : Block explorer url of Polygon Mumbai.

`string` object.

- `args`: During the execution of your function, These arguments are passed to the source code. The `args` value is `["1", "bitcoin", "btc-bitcoin"]`. These arguments are BTC IDs at CoinMarketCap, CoinGecko, and Coinpaprika. You can adapt args to fetch other asset prices.
- `secrets`: The secrets object that will be encrypted.
- `slotIdNumber`: Slot ID at the DON where to upload the encrypted secrets.
- `expirationTimeMinutes`: Expiration time in minutes of the encrypted secrets.
- `gasLimit`: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
- 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.

2. Simulating your request in a local sandbox environment:
   - Use `simulateScript` from the Chainlink Functions NPM package.
   - 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.uint256` in this example).
3. 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:
   - Initialize a `SecretsManager` instance from the Functions NPM package, then call the `encryptSecrets` function.
   - Call the `uploadEncryptedSecretsToDON` function of the `SecretsManager` instance. This function returns an object containing a `success` boolean as long as `version`, the secret version on the DON storage.
   - Call the `buildDONHostedEncryptedSecretsReference` function of the `SecretsManager` instance and use the slot ID and version to encode the DON-hosted encrypted secrets reference.
4. Encode the request data offchain using the `buildRequestCBOR` function from the Functions NPM package.

signer.

- Call the `updateRequest` function of your consumer contract.

readLatest.js

This explanation focuses on the readLatest script and that reads the latest receive response of your consumer contract then decode it offchain using the Chainlink Function NPM package.

The script has one hardcoded values that you have to change using your own Functions consumer contract address:

```
const consumerAddress = "0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA" // REPLACE this v yo
```

The primary function that the script executes is `readLatest`. This function can be broken into two main parts:

1. Read the latest response:
   - Initialize your functions consumer contract using the contract address, abi, and ethers provider.
   - Call the `s_lastRequestId`, `s_lastResponse`, and `s_lastError` functions of your consumer contract.
2. Decode the latest response:
   - If there was an error, read the latest error and parse it to `string`.
   - If there was no error, use the Functions NPM package `decodeResult` function and `ReturnType` enum to decode the response to the expected returned type (`ReturnType.uint256` in this example).

Stay updated on the latest Chainlink news

EN

Subscribe now

Developers

Developer resources

LINK Token Contracts

Data Feeds

VRF

Automation

Functions

CCIP

Solutions

Overview

DeFi

Chainlink VRF

Community

Chainlink Hackathon

Community overview

Grant program

Events

Become an advocate

EN

Chainlink

Ecosystem

Press

Blog

Team

Brand assets

FAQs

Contact

Security

Support

Press inquiries

Social

Twitter

YouTube

Discord

Telegram

WeChat

Reddit

EN