Automate your Functions (Time-based Automation)

This tutorial shows you how to useChainlink 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 theAPI multiple calls tutorial before you follow the steps in this example. This tutorial uses the same example but with an important difference:

- You will deployAutomatedFunctionsConsumerExample.sol instead of theFunctionsConsumerExample 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 theterms can result in suspension or termination of your Chainlink Functions account.

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 tutorial is configured to get the medianBTC/USDprice from multiple data sources according to a time schedule. For a detailed explanation of the code example, read theExamine the code section.

You can locate the scripts used in this tutorial in theexamples/10-automate-functionsdirectory .

1. Make sure to understand theAPI 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 theminimum balance for uploading encrypted secrets section to learn more). 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. This guide recommends maintaining at least 2 LINK within your subscription.
3. Get a free API key fromCoinMarketCap and note your API key.
4. Runnpx env-enc setto add an encryptedCOINMARKETCAP_API_KEYto your.env.encfile.

npx env-encset

Deploy an Automated Functions Consumer contract

1. Deploy a Functions consumer contract onPolygon Mumbai:
2. Open theAutomatedFunctionsConsumerExample.sol in Remix.

Open in Remix What is Remix?1. Compile the contract. 2. Open MetaMask and select thePolygon Mumbainetwork. 3. In Remix under theDeploy & Run Transactionstab, selectInjected Provider -

MetaMaskin theEnvironmentlist. Remix will use the MetaMask wallet to communicate withPolygon Mumbai. 4. Under theDeploysection, fill in the router address for your specific blockchain. You can find this address on theSupported Networks page. ForPolygon Mumbai, the router address is0x6E2dc0F9DB014aE19888F539E59285D2Ea04244C. 5. 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. 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

The consumer contract that you deployed is designed to be used with atime-basedautomation. Follow the instructions in theAutomation Job Scheduler guide to register your deployed contract using theChainlink 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 fromautomatedFunctions.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

There are two balances that you must monitor:

- Your subscription balance: Your balance will be charged each time your Chainlink Functions is fulfilled. If your balance is insufficient, your contract cannot send requests. Automating 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 theChainlink Functions Subscription Manager .
- Your upkeep balance: You can check this balance on theChainlink 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 thesendRequestCBORfunction. This security measure is important to prevent anyone from calling several timessendRequestCBORand draining your Functions subscription balance. Follow these steps:

2. On RemixIDE, under theDeploy & Transactionstab, locate your deployed Functions consumer contract.

3. Open the list of functions.
4. Fill in thesetAutomationCronContractfunction with the upkeep contract address you copied from the previous step.
5. Click ontransact. A Metamask popup appears and asks you to confirm the transaction.
6. Confirm the transaction and wait for it to be confirmed.

7. Configure the request details by calling theupdateRequestfunction. 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 (seeExamine the code). To do so, follow these steps:

8. On a terminal, go to theFunctions tutorialsdirectory .

9. OpenupdateRequest.js and replace the consumer contract address and the subscription ID with your own values:

constconsumerAddress="0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID1. Run theupdateRequest.js script.

nodeexamples/10-automate-functions/updateRequest.jsExample:

$ 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: '0x0000000000000000000000000000000000000000000000000000000000002737df' } ✅ Decoded response to uint256: 2570207n

Make request... Upload encrypted secret to gateways https://01.functions-gateway.testnet.chain.link/user. StorageSlotId 0. Expiration in minutes: 150

✅ Secrets uploaded properly to gateways https://01.functions-gateway.testnet.chain.link/user! Gateways response: { version: 1693925775, success: true }

✅ Automated Functions request settings updated! Transaction hash 0x6017e4b36f80c7d06e0f0f1dd48239e688419a8141b90ad944bff89232fa8e14 - Check the explorer https://mumbai.polygonscan.com/tx/0x6017e4b36f80c7d06e0f0f1dd48239e688419a8141b90ad944bff89232fa8e14

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 endpointhttps://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 theChainlink Automation App and connect to Polygon Mumbai. Your upkeep will be listed underMy upkeeps:

Click on your upkeep to fetch de details:

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

1. OpenreadLatest.jsand replace the consumer contract address with your own values:

constconsumerAddress="0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA"// REPLACE this with your Functions consumer address 1. Run thereadLatest script.

nodeexamples/10-automate-functions/readLatest.js 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 theChainlink 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: 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/
*** @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 */contractAutomatedFunctionsConsumerExampleisFunctionsClient,ConfirmedOwner{addresspublicupkeepContract;bytespublicrequest;uint64publicsubscriptionId;uint32publicgasLimit;by**
{}/ * @notice Reverts if called by anyone other than the contract owner or automation registry. */modifieronlyAllowed()*

{if(msg.sender!=owner()&&msg.sender!=upkeepContract)revertNotAllowedCaller(msg.sender,owner(),upkeepContract);_;}functionsetAutomationCronContract(address_upkeepContract)externalonlyOwn
@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 setfunctionupdateRequest(bytesmemory_request,uint64_subscriptionId,uint32_gasLimit,bytes32_donID)externalonlyOwner{request=_request;subscriptionId=_subscriptionId;gasLimit=_gasLimit;donID=_ @notice Send a pre-encoded CBOR request * @return requestId The ID of the sent request* /functionsendRequestCBOR()externalonlyAllowedreturns(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 an automated Chainlink Functions consumer contract, your contract must importFunctionsClient.sol . You can read the API reference ofFunctionsClient .

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/dev/v1_0_0/FunctionsClient.sol"; * TheupkeepContractaddress is stored in the contract storage. The contract owner sets this variable by calling thesetAutomationCronContractfunction.Note: This variable is used by theonlyAllowedto ensure only the upkeep contract can call thesendRequestCBORfunction.

address public upkeepContract * The encodedrequest,subscriptionId,gasLimit, andjobIdare stored in the contract storage. The contract owner sets these variables by calling theupdateRequestfunction.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 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 two remaining functions are:

- sendRequestCBORfor sending a request already encoded inbytes. It sends the request to the router by calling theFunctionsClientsendRequestfunction.
- 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);

source.js

The JavaScript code is similar to theCall Multiple Data Sources tutorial.

updateRequest.js

This explanation focuses on theupdate.js script and shows how to use theChainlink 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 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/automatedFunctions.json: The abi of the contract your script will interact with.Note: The script was tested with thisAutomatedFunctionsConsumer contract .

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

constconsumerAddress="0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID
The primary function that the script executes isupdateRequestMumbai. 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. gatewayUrls: The secrets endpoint URL to which you will upload the encrypted secrets.
5. explorerUrl: Block explorer url of Polygon Mumbai.
6. 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.
7. args: During the execution of your function, These arguments are passed to the source code. Theargsvalue is["1", "bitcoin", "btc-bitcoin"]. These arguments are BTC IDs at CoinMarketCap, CoinGecko, and Coinpaprika. You can adapt args to fetch other asset prices.
8. secrets: The secrets object that will be encrypted.
9. slotIdNumber: Slot ID at the DON where to upload the encrypted secrets.
10. expirationTimeMinutes: Expiration time in minutes of the encrypted secrets.
11. gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
12. Initialization of etherssignerandproviderobjects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.
13. Simulating your request in a local sandbox environment:
14. UsesimulateScriptfrom the Chainlink Functions NPM package.
15. Read theresponseof the simulation. If successful, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.uint256in this example).
16. 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:
17. Initialize aSecretsManagerinstance from the Functions NPM package, then call theencryptSecretsfunction.
18. Call theuploadEncryptedSecretsToDONfunction of theSecretsManagerinstance. This function returns an object containing asuccessboolean as long asversion, the secret version on the DON storage.
19. Call thebuildDONHostedEncryptedSecretsReferencefunction of theSecretsManagerinstance and use the slot ID and version to encode the DON-hosted encrypted secrets reference.
20. Encode the request data offchain using thebuildRequestCBORfunction from the Functions NPM package.
21. Update the Functions consumer contract:
22. Initialize your functions consumer contract using the contract address, abi, and ethers signer.
23. Call theupdateRequestfunction of your consumer contract.

readLatest.js

This explanation focuses on thereadLatest 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:

constconsumerAddress="0x5abE77Ba2aE8918bfD96e2e382d5f213f10D39fA"// REPLACE this with your Functions consumer address The primary function that the script executes isreadLatest. This function can be broken into two main parts:

1. Read the latest response:
2. Initialize your functions consumer contract using the contract address, abi, and ethers provider.
3. Call thes_lastRequestId,s_lastResponse, ands_lastErrorfunctions of your consumer contract.
4. Decode the latest response:
5. If there was an error, read the latest error and parse it tostring.
6. If there was no error, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.uint256in this example).