Back

Automate your Functions (Custom Logic 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 Automate your Functions (Time-based Automation) tutorial before you follow the steps in this example. This tutorial explains how to trigger your functions using an Automation compatible contract.

After you deploy and set up your contract, Chainlink Automation triggers your function on every block.

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.

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.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
```

EN

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.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.

EN

2. Compile the contract.
3. Open MetaMask and select the *Polygon Mumbai* network.
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 on every block. Read the Examine the code section for a detailed explanation of the code example.

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

Subscription guide. This guide recommends maintaining at least 2 LINK within your subscription.

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

> **CAUTION**
>
> When using Chainlink Automation, developers should be mindful of the risks of Automation attempting to perform upkeeps indefinitely if a previous upkeep fails to update due to reversion. To learn more, read the Automation best practices.

1. Deploy a Functions consumer contract on *Polygon Mumbai*:
   1. Open the CustomAutomatedFunctionsConsumerExample.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.

existing subscription.

Configure your Automation Consumer contract

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
```

3. Run the updateRequest.js script to update your Functions consumer contract's request details.

Configure Chainlink Automation

The consumer contract that you deployed is designed to be used with a **custom logic** automation. Follow the instructions in the Register a Custom Logic Upkeep guide to register your deployed contract using the Chainlink Automation App. Use the following upkeep settings:

- Trigger: Custom logic
- Target contract address: The address of the Chainlink Functions consumer contract that you deployed
- Check data: Leave this field blank
- Gas limit: `1000000`
- Starting balance (LINK): `1`

You can leave the other settings at their default values for the example in this tutorial.

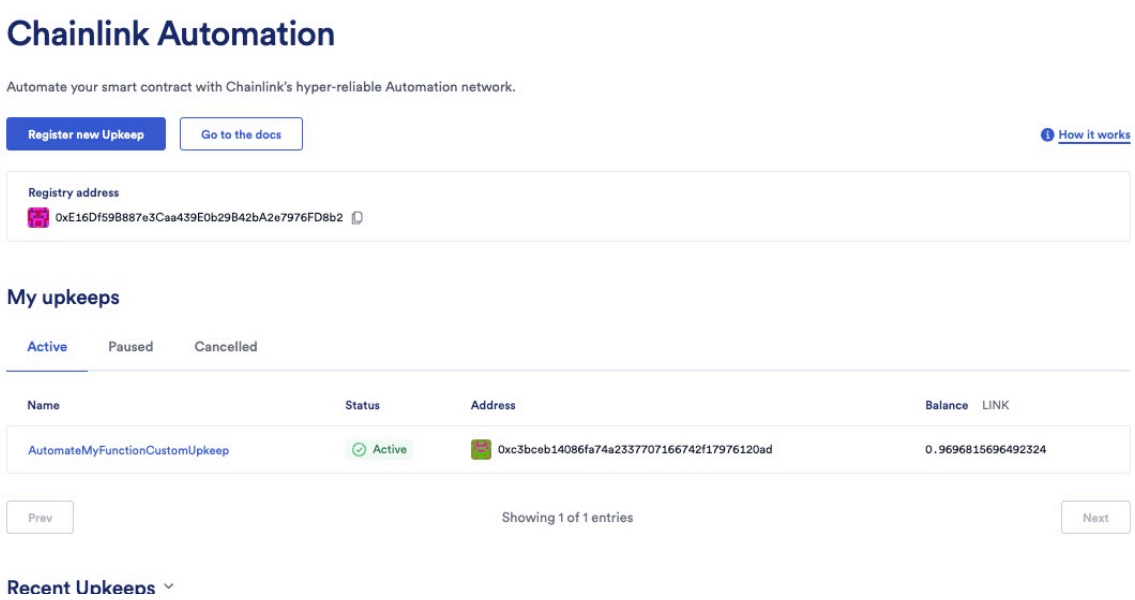
MONITOR YOUR BALANCES

There are two balances that you must monitor:

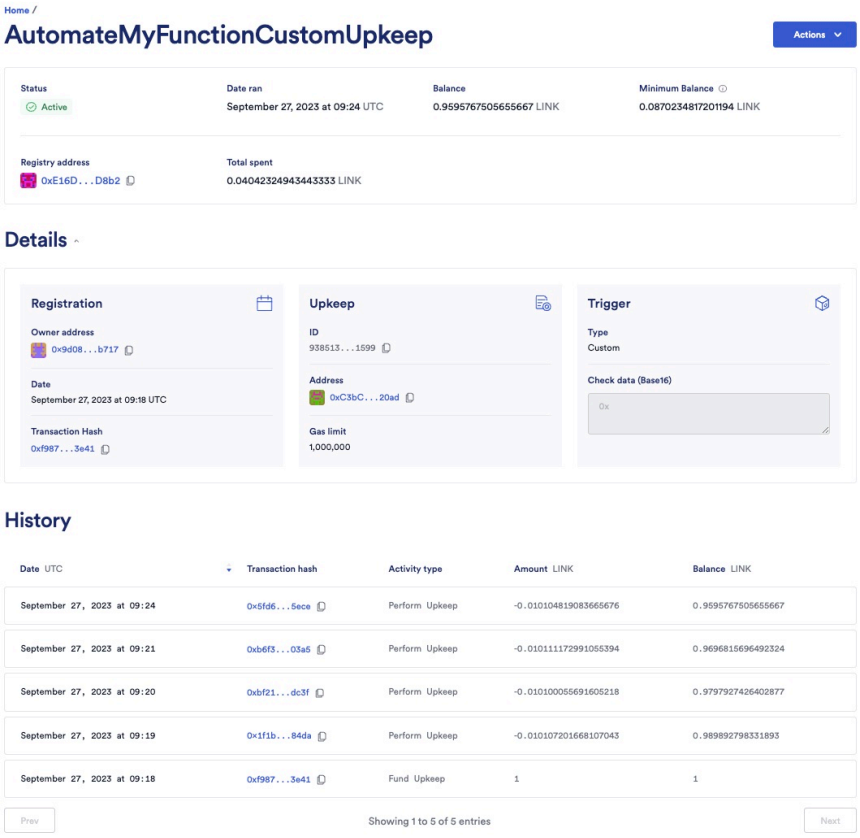
- Your subscription balance: Your balance will be charged each time your Chainlink Functions is fulfilled. If your balance is insufficient, your contract cannot send requests. Automating your Chainlink Functions means they will be regularly triggered, so monitor and fund your subscription account regularly. You can check your subscription details (including the balance in LINK) in 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.

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:



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
```

1. Run the readLatest script.

```
node examples/10-automate-functions/readLatest.js
```

Example:

```
$ node examples/10-automate-functions/readLatest.js
secp256k1 unavailable, reverting to browser version
Last request ID is 0x310d57a7af34ae4ce565f5745ff46fe2706e96b25b3172ada60cc60f4603b38e
✅ Decoded response to uint256:  2625865n
```

Clean up

After you finish the guide, cancel your upkeep in the Chainlink Automation App and

EN

Examine the code

CustomAutomatedFunctionsConsumer.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 {AutomationCompatibleInterface} from "@chainlink/contracts/src/v0.8/automation/Automatic
import {ConfirmedOwner} from "@chainlink/contracts/src/v0.8/shared/access/ConfirmedOwner.sol";
import {FunctionsRequest} from "@chainlink/contracts/src/v0.8/functions/v1_0_0/libraries/Functionsl

/**
 * @title Functions contract used for Automation.
 * @notice This contract is a demonstration of using Functions and Automation.
 * @notice NOT FOR PRODUCTION USE
 */
contract CustomAutomatedFunctionsConsumerExample is
    FunctionsClient,
    AutomationCompatibleInterface,
    ConfirmedOwner
{
    uint256 public lastBlockNumber;
    bytes public request;
    uint64 public subscriptionId;
    uint32 public gasLimit;
    bytes32 public donID;
    bytes32 public s_lastRequestId;
    bytes public s_lastResponse;
    bytes public s_lastError;
    uint256 public s_upkeepCounter;
    uint256 public s_requestCounter;
    uint256 public s_responseCounter;

    error UnexpectedRequestID(bytes32 requestId);

    event Response(bytes32 indexed requestId, bytes response, bytes err);
    event RequestRevertedWithErrorMsg(string reason);
    event RequestRevertedWithoutErrorMsg(bytes data);

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

EN

```
* @dev This function checks if the current block number has incremented since the last recorded blocl
* @return upkeepNeeded A boolean indicating if upkeep is needed (true if the current block number h
* @return performData An empty bytes value since no additional data is needed for the upkeep in this
*/
  function checkUpkeep(
      bytes calldata /* checkData */
  )
      external
      view
      override
      returns (bool upkeepNeeded, bytes memory performData)
  {
      upkeepNeeded = block.number - lastBlockNumber > 0; // Check if the current block numbe
      // We don't use the checkData in this example. The checkData is defined when the Upkeep was
      return (upkeepNeeded, ""); // Return an empty bytes value for performData
  }

  /**
* @notice Send a pre-encoded CBOR request if the current block number has incremented since the la
*/
  function performUpkeep(bytes calldata /* performData */) external override {
      if (block.number - lastBlockNumber > 0) {
          lastBlockNumber = block.number;
          s_upkeepCounter = s_upkeepCounter + 1;
          try
              i_router.sendRequest(
                  subscriptionId,
                  request,
                  FunctionsRequest.REQUEST_DATA_VERSION,
                  gasLimit,
                  donID
              )
          returns (bytes32 requestId) {
              s_lastRequestId = requestId;
              s_requestCounter = s_requestCounter + 1;
              emit RequestSent(requestId);
          } catch Error(string memory reason) {
              emit RequestRevertedWithErrorMsg(reason);
          } catch (bytes memory data) {
              emit RequestRevertedWithoutErrorMsg(data);
          }
      }
      // We don't use the performData in this example. The performData is generated by the Automat
  }

  /// @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
```

EN

```
    function updateRequest(
        bytes memory _request,
        uint64 _subscriptionId,
        uint32 _gasLimit,
        bytes32 _donID
    ) external onlyOwner {
        request = _request;
        subscriptionId = _subscriptionId;
        gasLimit = _gasLimit;
        donID = _donID;
    }

    /**
   * @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;
        s_responseCounter = s_responseCounter + 1;
        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 or
```

- To write a compatible Automations contract, your contract must import

```
import {AutomationCompatibleInterface} from "@chainlink/contracts/src/v0.8/autom
```

- The `lastBlockNumber` is stored in the contract. It represents the block number of the last time `performUpkeep` was triggered.
- 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);
```

- We define two events that your smart contract emits when sending a request to Chainlink Functions fails:

```
event RequestRevertedWithErrorMsg(string reason);
event RequestRevertedWithoutErrorMsg(bytes data);
```

- Pass the router address for your network when you deploy the contract:

```
constructor(address router) FunctionsClient(router)
```

- The three remaining functions are:
  - `checkUpkeep` for checking offchain if `performUpkeep` should be executed. `performUpkeep` should only be executed if the current block number is higher than the block number of the last execution.
  - `performUpkeep`: Executed by Chainlink Automation when `checkUpkeep` returns `true`. This function sends the request (encoded in `bytes`) to the router by calling the `FunctionsClient sendRequest` function. **Note**: We use *try and catch* to gracefully handle reverts of `i_router.sendRequest` by emitting an event. We also update the

- `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`, then increments the response counter `s_responseCounter` before emitting the `Response` event.

```
s_lastResponse = response;
s_lastError = err;
s_responseCounter = s_responseCounter + 1;
emit Response(requestId, s_lastResponse, s_lastError);
```

source.js

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

updateRequest.js

The JavaScript code is similar to the one used in the Automate your Functions (Time-based Automation) tutorial.

readLatest.js

The JavaScript code is similar to the one used in the Automate your Functions (Time-based Automation) tutorial.

Blog

Team

Brand assets

FAQs

Contact

Security

Support

Press inquiries

Social

Twitter

YouTube

Discord

Telegram

WeChat

Reddit

Privacy Policy Terms of Use