

Create Automation-Compatible Contracts

Learn how to make smart contracts that are compatible with Automation.

Considerations and Best Practices

Before you deploy contracts to use with Chainlink Automation, read the [Best Practices](#) guide. These best practices are important for using Chainlink Automation securely and reliably. You can also read more about the Chainlink Automation architecture [here](#).

Automation compatible contracts

A contract is Automation-compatible when it follows a specified interface that allows the Chainlink Automation Network to determine if, when, and how the contract should be automated.

The interface you use will depend on the type of trigger you want to use:

- If you want a log event to trigger your upkeep, use the [LogAutomation](#) interface.
- If you want to use onchain state in a custom calculation to trigger your upkeep, use [AutomationCompatibleInterface](#) interface.
- If you want to call a function just based on time, you don't need an interface. Consider instead using [time-based upkeep](#).
- If you want to use Automation with Data Streams, use [StreamsLookupCompatibleInterface](#) interface.

You can learn more about these interfaces [here](#).

Example Automation-compatible contract using custom logic trigger

Custom logic Automation compatible contracts must meet the following requirements:

- Import `AutomationCompatible.sol`. You can refer to the [Chainlink Contracts](#) on GitHub to find the latest version.
- Use the [AutomationCompatibleInterface](#) from the library to ensure your `checkUpkeep` and `performUpkeep` function definitions match the definitions expected by the Chainlink Automation Network.
- Include a `checkUpkeep` function that contains the logic that will be executed offchain to see if `performUpkeep` should be executed. `checkUpkeep` can use onchain data and a specified `checkData` parameter to perform complex calculations offchain and then send the result to `performUpkeep` as `performData`.
- Include a `performUpkeep` function that will be executed onchain when `checkUpkeep` returns true.

Use these elements to create a compatible contract that will automatically increment a counter after every `updateInterval` seconds. After you register the contract as an upkeep, the Chainlink Automation Network frequently simulates your `checkUpkeep` offchain to determine if the `updateInterval` time has passed since the last increment (timestamp). When `checkUpkeep` returns true, the Chainlink Automation Network calls `performUpkeep` onchain and increments the counter. This cycle repeats until the upkeep is cancelled or runs out of funding.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

// AutomationCompatible.sol imports the functions from both ./AutomationBase.sol and
./interfaces/AutomationCompatibleInterface.sol
import {AutomationCompatibleInterface} from "@chainlink/contracts/src/v0.8/automation/AutomationCompatible.sol";

* @dev Example contract, use the Forwarder as needed for additional security.
* @notice important to implement {AutomationCompatibleInterface}
* // 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 Counter is AutomationCompatibleInterface {
    * Public counter variable
    uint256 public counter;
    * Use an interval in seconds and a timestamp to slow execution of Upkeep
    uint256 public immutable interval;
    uint256 public lastTimeStamp;
    constructor(uint256 updateInterval) {
        interval = updateInterval;
        lastTimeStamp = block.timestamp;
        counter = 0;
    }
    function checkUpkeep(bytes calldata checkData)
    /external view override returns (bool upkeepNeeded, bytes memory performData) {
        upkeepNeeded = (block.timestamp - lastTimeStamp) > interval;
        // We don't use the checkData in this example. The checkData is defined when the Upkeep was registered.
        function performUpkeep(bytes calldata performData)
        */
        external override {
            if ((block.timestamp - lastTimeStamp) > interval) {
                lastTimeStamp = block.timestamp;
                counter = counter + 1;
            }
            // We don't use the performData in this example.
            // The performData is generated by the Automation Node's call to your checkUpkeep function.
        }
    }
}

Open in Remix What is Remix?
Compile and deploy your own Automation Counter onto a supported Testnet.
```

1. In the Remix example, select the compile tab on the left and press the compile button. Make sure that your contract compiles without any errors. Note that the Warning messages in this example are acceptable and will not block the deployment.
2. Select the Deploy tab and deploy the `CounterSmart` contract in the injected web3 environment. When deploying the contract, specify the `updateInterval` value. For this example, set a short interval of 60. This is the interval at which the `performUpkeep` function will be called.
3. After deployment is complete, copy the address of the deployed contract. This address is required to register your upkeep in the [Automation UI](#). The example in this document uses [custom logic](#) automation.

To see more complex examples, go to the [Quick Starts](#) page.

Now register your [upkeep](#).

Vyper example

Note on arrays

Make sure the `checkdata` array size is correct. Vyper does not support dynamic arrays.

You can find a `KeepersConsumer` example [here](#). Read the `apeworx-starter-kit` [README](#) to learn how to run the example.