**Implementing Chainlink Automation**

**Implementing Chainlink Automation**

Remember how Richard deleted the performUpkeep and checkUpkeep in the previous videos ... we gonna need those if we want to use the Chainlink Automation without interacting with Chainlink's front-end. We are engineers, we do not use front-ends!

For this to work we need to refactor the pickWinner function. That functionality needs to be part of the performUpkeep if we want the Chainlink node to call it for us. But before that, let's create the checkUpkeep function:

/\*\*

\* @dev This is the function that the Chainlink Keeper nodes call

\* they look for `upkeepNeeded` to return True.

\* the following should be true for this to return true:

\* 1. The time interval has passed between raffle runs.

\* 2. The lottery is open.

\* 3. The contract has ETH.

\* 4. There are players registered.

\* 5. Implicitly, your subscription is funded with LINK.

\*/

function checkUpkeep(bytes memory /\* checkData \*/) public view returns (bool upkeepNeeded, bytes memory /\* performData \*/) {

bool isOpen = RaffleState.OPEN == s\_raffleState;

bool timePassed = ((block.timestamp - s\_lastTimeStamp) >= i\_interval);

bool hasPlayers = s\_players.length > 0;

bool hasBalance = address(this).balance > 0;

upkeepNeeded = (timePassed && isOpen && hasBalance && hasPlayers);

return (upkeepNeeded, "0x0");

}

Again, a lot of things up there, but fear not, we are going to explain everything.

Going back to the [Chainlink Automation tutorial](https://docs.chain.link/chainlink-automation/guides/compatible-contracts) we see that checkUpkeep can use onchain data and a specified checkData parameter to perform complex calculations offchain and then send the result to performUpkeep as performData. But in our case, we don't need that checkData parameter. If a function expects an input, but we are not going to use it we can comment it out like this: /\* checkData \*/. Ok, moving on, we'll make checkUpkeep public view and we match the expected returns of (bool upkeepNeeded, bytes memory /\* performData \*/) commenting out that performData because we aren't going to use it.

You are amazing! Keep going!

Back to our raffle now, what are the conditions required to be true in order to commence the winner-picking process? We've placed the answer to this in the NATSPEC comments.

\* 1. The time interval has passed between raffle runs.

\* 2. The lottery is open.

\* 3. The contract has ETH.

\* 4. There are players registered.

\* 5. Implicitly, your subscription is funded with LINK.

For points 1-3 we coded the following lines:

bool isOpen = RaffleState.OPEN == s\_raffleState;

bool timePassed = ((block.timestamp - s\_lastTimeStamp) > i\_interval);

bool hasPlayers = s\_players.length > 0;

bool hasBalance = address(this).balance > 0;

We check if the Raffle is in the open state, if enough time has passed and if there are players registered to the Raffle and if we have a prize to give out. All these need to be true for the winner-picking process to be able to run.

In the end, we return the two elements required by the function declaration:

return (upkeepNeeded, "0x0");

"0x0" is just 0 in bytes, we ain't going to use this return anyway.

Amazing!

Chainlink nodes will call this checkUpkeep function. If the return upkeepNeeded is true, then they will call performUpkeep ... which in our case is the pickWinner function. Let's refactor it a little bit:

// 1. Get a random number

// 2. Use the random number to pick a player

// 3. Automatically called

function performUpkeep(bytes calldata /\* performData \*/) external override {

(bool upkeepNeeded, ) = checkUpkeep("");

// require(upkeepNeeded, "Upkeep not needed");

if (!upkeepNeeded) {

revert Raffle\_\_UpkeepNotNeeded(

address(this).balance,

s\_players.length,

uint256(s\_raffleState)

);

}

s\_raffleState = RaffleState.CALCULATING;

uint256 requestId = i\_vrfCoordinator.requestRandomWords(

i\_gasLane,

i\_subscriptionId,

REQUEST\_CONFIRMATIONS,

i\_callbackGasLimit,

NUM\_WORDS

);

}

We copied the start from the [Chainlink Automation tutorial](https://docs.chain.link/chainlink-automation/guides/compatible-contracts" \t "_blank) renaming the pickWinner function. Given that our new performUpkeep is external, as it should be if we want one of the Chainlink nodes to call it, we need to ensure that the same conditions are required for everyone else to call it. In other words, there are two possibilities for this function to be called:

1. A Chainlink node calls it, but it will first call checkUpkeep, see if it returns true, and then call performUpkeep.
2. Everyone else calls it ... but nothing is checked.

We need to fix point 2.

For that we will make the function perform a call to checkUpkeep:

(bool upkeepNeeded, ) = checkUpkeep("");

And we check it's result. If the result is false we revert with a new custom error:

if (!upkeepNeeded) {

revert Raffle\_\_UpkeepNotNeeded(

address(this).balance,

s\_players.length,

uint256(s\_raffleState)

);

}

Let's define it at the top of the contract, next to the other errors:

error Raffle\_\_UpkeepNotNeeded(

uint256 currentBalance,

uint256 numPlayers,

uint256 raffleState

);

This is the first time when we provided some parameters to the error. Think about them as extra info you get when you receive the error.

**Note: you can do both**uint256 raffleState**or**RaffleState raffleState**because these are directly convertible.**

We leave the rest of the function intact.

Another thing that we should do is to import the AutomationCompatibleInterface:

import {AutomationCompatibleInterface} from "chainlink/src/v0.8/automation/interfaces/AutomationCompatibleInterface.sol";

and let's make our contract inherit it:

contract Raffle is VRFConsumerBaseV2, AutomationCompatibleInterface {

Now let's call a forge build to see if everything is ok.

Amazing work!