Smart Contract Resolvers

Here is how a simple smart contract resolver looks like. This resolver is responsible for checking thelastExecuted state of a simple counter in another smart contract. And if 3 minutes have passed since the last execution, this resolver will prompt Gelato to execute.

Check out the code forCounter andCounterResolver .

```

CounterResolver.sol Copy contractCounterResolver{ ICounterpublicimmutablecounter;

constructor(ICounter_counter) { counter=_counter; }

functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) {
uint256lastExecuted=counter.lastExecuted();

canExec=(block.timestamp-lastExecuted)>180;

execPayload=abi.encodeCall(ICounter.increaseCount,(1)); } }

```

A resolver should always return 2 things:

1. boolcanExec
2. : whether Gelato should execute the task.
3. bytesexecData
4. : data that executors should use for the execution.
5.

Resolvers can also accept arguments. This is useful as you can potentially "re-use" your resolver when creating multiple tasks.

Using the same example as above:

```

Copy functionchecker(address_counter) external view returns(boolcanExec,bytesmemoryexecPayload) {
uint256lastExecuted=ICounter(_counter).lastExecuted();

canExec=(block.timestamp-lastExecuted)>180;

execPayload=abi.encodeCall(ICounter.increaseCount,(1)); }

```

Instead of a hardcodedCOUNTER address, you can passcounterAddress as an argument.

Common patterns and best practices

Checking multiple functions in one Resolver

Let's say you want to automate your yield compounding for multiple pools. To avoid creating multiple tasks and having different resolvers, you could keep a list of the pools and iterate through each of the pools to see if they should be compounded. Take a look at this example.

```

Copy functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) { uint256delay=harvester.delay();

for(uint256i=0; i<vaults.length(); i++) { IVault vault=IVault(getVault(i));

canExec=block.timestamp>=vault.lastDistribution().add(delay);

execPayload=abi.encodeWithSelector( IHarvester.harvestVault.selector, address(vault) );

if(canExec)return(true,execPayload);

}

return(false,bytes("No vaults to harvest");
```

}

```

This resolver will return true when a certain time has elapsed since the last distribution of a pool, together with the payload to compound that specific pool.

Logs using custom return messages

The Gelato Automate UI has a feature which provides logs which show what the Gelato Executors are seeing in real time.

Using custom return messages, you can always check where in your smart contract Resolver the logic is currently "stuck", i.e. why isn't the Resolver returningtrue .

```

Copy functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) {
uint256lastExecuted=counter.lastExecuted();

if(block.timestamp-lastExecuted<180)return(false,bytes("Time not elapsed");

execPayload=abi.encodeCall(ICounter.increaseCount,(1)); return(true,execPayload); }

```

Limit the Gas Price of your execution

On networks such as Ethereum, gas will get expensive at certain times. If what you are automating is not time-sensitive and don't mind having your transaction mined at a later point, you can limit the gas price used in your execution in your resolver.

```

Copy functionchecker() external view returns(boolcanExec,bytesmemoryexecPayload) { // condition here

if(tx.gasprice>80gwei)return(false,bytes("Gas price too high")); }

```

This way, Gelato will not execute your transaction if the gas price is higher than 80 GWEI.

Previous Custom logic triggers Next Web3 Functions Last updated4 months ago On this page *Common patterns and best practices * Checking multiple functions in one Resolver * Logs using custom return messages * Limit the Gas Price of your execution
```