

# How to use oracles in Arbitrum app

Oracles are web services that provide a connection between smart contracts and the outside world. They let decentralized apps (dApps) interact with off-chain data and services.

Familiarity with [oracles](#), smart contracts, and blockchain development is expected. If you're new to blockchain development, consider reviewing our [Quickstart: Build a dApp with Arbitrum \(Solidity, Hardhat\)](#) before proceeding.

## Chainlink

[Chainlink](#) is a widely-recognized Web3 services platform that specializes in decentralized oracle networks. It lets you build Ethereum and Arbitrum dApps that connect to a variety of off-chain data feeds and APIs, including those that provide asset prices, weather data, random number generation, and more.

### Querying the price of ARB through Chainlink

Here's an example on how to use a price feed from Chainlink to query the current price of ARB on-chain. We'll use an interface provided by Chainlink that can be configured with the address of the proxy that holds the information we want to request, and wrap the operation in a contract.

Chainlink provides an npm package with the contracts needed to access their feeds. We first install that package in our project:

yarn add @chainlink/contracts To use a data feed, we retrieve the information through the `AggregatorV3Interface` and the proxy address of the feed we want to query.

```
import
```

```
"@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol" ; In this case, we want to obtain the current price of ARB in USD in Arbitrum One, so we need to know the address of the proxy that will provide that information. Chainlink maintains a list of price feed address here . For ARBUSDT, we'll use the address 0xb2A824043730FE05F3DA2efaFa1CBbe83fa548D6 .
```

We can now build the function to get the latest price of ARB. We'll use this example contract:

```
contract
```

```
ARBPriceConsumer
```

```
{ AggregatorV3Interface internal priceFeed ;
```

```
/* * Network: Arbitrum One * Aggregator: ARB/USD * Address: 0xb2A824043730FE05F3DA2efaFa1CBbe83fa548D6 address
```

```
constant PROXY =
```

```
0xb2A824043730FE05F3DA2efaFa1CBbe83fa548D6 ;
```

```
constructor ( )
```

```
{ priceFeed =
```

```
AggregatorV3Interface ( PROXY ) ; }
```

```
/* * Returns the latest price.*/ function
```

```
getLatestPrice ( )
```

```
public
```

```
view
```

```
returns
```

```
( int )
```

```
{ ( / uint80 roundID / , int price , uint startedAt , uint timeStamp , uint80 answeredInRound / )
```

```
= priceFeed . latestRoundData ( ) ; return price ; } } You can adapt this contract to your needs. Just remember to use the address of the asset you want to request the price for in the appropriate network, and to deploy your contract to the same
```

network . Remember we have a [Quickstart](#) available that goes through the process of compiling and deploying a contract.

## More examples

Refer to [Chainlink's documentation](#) for more examples of querying price feeds plus other data feeds available.

## API3

[API3](#) is a collaborative project to deliver traditional API services to smart contract platforms in a decentralized and trust-minimized way. API3 provides the technology for Airnodes to push off-chain data to on-chain contracts. This data can then be queried directly through the Airnode (initiating a "pull-type" request) or through dAPIs (data feeds of up-to-date off-chain data).

### Querying the price of ARB through API3

Here's an example on how to use an API3 data feed to query the current price of ARB on-chain. The [API3 market](#) provides a list of all the dAPIs available across multiple chains including testnets. These dAPIs are self-funded so, before querying it, we must make sure they have enough funds to cover our test.

API3 provides an npm package with the contracts needed to access their feeds. We first install that package in our project:

```
yarn  
  
add @api3/contracts To use a data feed, we retrieve the information through the specific proxy address for that feed. We'll use the IProxy interface to do so.
```

```
import
```

```
"@api3/contracts/v0.8/interfaces/IProxy.sol" ; In this case, we want to obtain the current price of ARB in USD in Arbitrum One, so we need to know the address of the proxy that will provide that information. We will search the feed on the API3 Market, connect our wallet and click on Get Proxy . The ARB/USD proxy address is 0x0cB281EC7DFB8497d07196Dc0f86D2eFD21066A5 .
```

We can now build the function to get the latest price of ARB. We'll use this example contract:

```
contract
```

```
ARBPriceConsumer
```

```
{ /* * Network: Arbitrum One * Aggregator: ARB/USD * Proxy: 0x0cB281EC7DFB8497d07196Dc0f86D2eFD21066A5  
address
```

```
constant PROXY =
```

```
0x0cB281EC7DFB8497d07196Dc0f86D2eFD21066A5 ;
```

```
/* * Returns the latest price.*/ function
```

```
getLatestPrice ( ) external view returns
```

```
( int224 value ,
```

```
uint256 timestamp ) { ( value , timestamp )
```

```
=
```

```
IProxy ( PROXY ) . read ( ) ; // If you have any assumptions about value and timestamp, make sure // to validate them right after reading from the proxy. } } You can adapt this contract to your needs. Just remember to use the address of the asset you want to request the price for in the appropriate network, and to deploy your contract to the same network . Remember we have a Quickstart available that goes through the process of compiling and deploying a contract.
```

### Querying a random number through API3

[API3 QRNG](#) is a public utility provided with the courtesy of [Australian National University \(ANU\)](#) . It is served as a public good, it is free of charge (apart from the gas costs), and it provides quantum randomness when requiring RNG on-chain.

To request randomness on-chain, the requester submits a request for a random number to `AirnodeRrpV0` . The ANU Airnode gathers the request from the `AirnodeRrpV0` protocol contract, retrieves the random number off-chain, and sends it back to `AirnodeRrpV0` . Once received, it performs a callback to the requester with the random number.

Here's an example of a basicQrngRequester that requests a random number.

API3 provides an npm package with the contracts needed to access the ANU qrng airnode. We first install that package in our project:

```
yarn
```

add @api3/airnode-protocol We'll need several information to request a random number:

- address airnodeRrp
- : Address of the protocol contract. See the [Chains](#)
- page for a list of addresses on different chains. For Arbitrum, we'll use 0xb015ACeEdD478fc497A798Ab45fcED8BdEd08924
- .
- address airnode
- : The address that belongs to the Airnode that will be called to get the QRNG data via its endpoints. See the [Providers](#)
- page for a list of addresses on different chains. For Arbitrum we'll use 0x9d3C147cA16DB954873A498e0af5852AB39139f2
- .
- bytes32 endpointId
- : Endpoint ID known by the Airnode that will map to an API provider call (allowed to be bytes32(0)
- ). You can also find that information in the [Providers](#)
- page. For Arbitrum we'll use 0xfb6d017bb87991b7495f563db3c8cf59ff87b09781947bb1e417006ad7f55a78
- .

We can now build the function to get a random number. We'll use this example contract:

```
import
```

```
"@api3/airnode-protocol/contracts/rrp/requesters/RrpRequesterV0.sol" ;
```

```
contract
```

```
QrngRequester
```

```
is RrpRequesterV0 { event
```

```
RequestedUint256 ( bytes32
```

```
indexed requestId ) ; event
```

```
ReceivedUint256 ( bytes32
```

```
indexed requestId ,
```

```
uint256 response ) ;
```

```
/* * Network: Arbitrum One * AirnodeRrpV0 Address: 0xb015ACeEdD478fc497A798Ab45fcED8BdEd08924 * Airnode: 0x9d3C147cA16DB954873A498e0af5852AB39139f2 * Endpoint ID: 0xfb6d017bb87991b7495f563db3c8cf59ff87b09781947bb1e417006ad7f55a78 / address
```

```
constant _airnodeRrp =
```

```
0xb015ACeEdD478fc497A798Ab45fcED8BdEd08924 ; address
```

```
constant airnode =
```

```
0x9d3C147cA16DB954873A498e0af5852AB39139f2 ; bytes32
```

```
constant endpointIdUint256 =
```

```
0xfb6d017bb87991b7495f563db3c8cf59ff87b09781947bb1e417006ad7f55a78 ; mapping ( bytes32
```

```
=>
```

```
bool )
```

```
public waitingFulfillment ;
```

```
constructor ( )
```

```
RrpRequesterV0 ( _airnodeRrp )
```

```

{ }

function

makeRequestUint256 ( )

external

{ bytes32 requestId = airnodeRrp . makeFullRequest ( airnode , endpointIdUint256 , address ( this ) , msg . sender , address
( this ) , this . fulfillUint256 . selector , "" ) ; waitingFulfillment [ requestId ]

=

true ; emit

RequestedUint256 ( requestId ) ; }

function

fulfillUint256 ( bytes32 requestId ,

bytes

calldata data ) external onlyAirnodeRrp { require ( waitingFulfillment [ requestId ] , "Request ID not known" ) ;
waitingFulfillment [ requestId ]

=

false ; uint256 qrngUint256 = abi . decode ( data ,

( uint256 ) ) ;

// Use qrngUint256 here...

emit

ReceivedUint256 ( requestId , qrngUint256 ) ; } } You can adapt this contract to your needs. Just remember to use the
addresses of the appropriate network, and to deploy your contract to the same network . Remember we have aQuickstart
available that goes through the process of compiling and deploying a contract.

```

## More examples

Refer to [API3's documentation](#) for more examples of querying other data feeds and Airnodes.

## Tellor

[Tellor](#) is a decentralized oracle network that incentivizes an open, permissionless network of data reporting and validation, ensuring that any verifiable data can be brought on-chain. It supports basic spot prices, sophisticated pricing specs (TWAP/VWAP), Snapshot Vote Results, and custom data needs.

## Querying the price of ETH through Tellor

Here's an example on how to use a Tellor data feed to query the current price of ETH on-chain. The way it works is that a query is crafted asking for the price of one currency against another and sent to the oracle contract. If the information for that query is available, it will be returned. Oracle contracts can be found in the [Contracts Reference](#) page.

Tellor provides an npm package with the contracts needed to query the contract. We first install that package in our project:

```

npm

install usingtellor Our function will just wrap the call to the oracle contract with the query we are interested in. In this case we
want to obtain the "SpotPrice" of "eth" against "usd". We will request this information to the Arbitrum oracle
contract0xD9157453E2668B2fc45b7A803D3FEF3642430cC0 . We'll use this example contract:

contract

ARBPriceConsumer

is UsingTellor { /* * Network: Arbitrum One * Aggregator: ARB/USD * Address:
0xD9157453E2668B2fc45b7A803D3FEF3642430cC0 / constructor ( address

```

```

payable _tellorAddress )

UsingTellor ( _tellorAddress ) { }

/* * Returns the latest price.*/ function

getLatestPrice ( )

public

view

returns

( uint256 )

{ bytes

memory _queryData = abi . encode ( "SpotPrice" , abi . encode ( "eth" ,

"usd" ) ) ; bytes32 _queryId =

keccak256 ( _queryData ) ;

( bytes

memory _value ,

uint256 _timestampRetrieved )

= getDataBefore ( _queryId , block . timestamp -

20 minutes ) ; if

( _timestampRetrieved ==

0 )

return

0 ; require ( block . timestamp - _timestampRetrieved <

24 hours ) ; return abi . decode ( _value ,

( uint256 ) ) ; } } You can adapt this contract to your needs. Just remember to use the ticker of the assets you want to
request the price for, and to deploy your contract to the appropriate network, with the address of the oracle contract in that
network . Remember we have a Quickstart available that goes through the process of compiling and deploying a contract.

```

## See also

- [Tellor's documentation](#)
- demonstrates how to query price feeds and other data feeds.
- [How to use Supra's price feed oracle](#)
- [How to use Supra's VRF](#) [Edit this page](#) Last updated on Mar 22, 2024 [Previous Oracles overview](#) [Next Oracles reference](#)