Data Asserter

Using the Optimistic Oracle V3 to assert arbitrary off-chain data on-chain. This section covers theData Asserter contract , which is available in the Optimistic Oracle V3quick-start repo . This tutorial shows an example of how to build an arbitrary data asserter contract, using the UMAOptimistic Oracle V3 (OOV3) contract. This enables you to assert to the correctness of some off-chain process and store the output on-chain. This tutorial acts as a starting point to show how the OOV3 can be flexibility used in a sample integration.

Data Asserter

This smart contract allows data providers to assert the correctness of a data point, within any arbitrary data set, and bring the result on-chain. This contract is designed to be maximally open ended, enabling any kind of data to be asserted. For example, and for illustrative purposes of this tutorial, we will assert weather data on-chain. Note, though, that this kind of assertion flow and the associated OOV3 integration could build a wide range of assertions with arbitrary complexity. For example you could use the same structure to bring on-chain complex data associated with Beacon chain validator set to build a Liquid staking derivative, enable complex off-chain computation with on-chain outputs or anything that has a defined deterministic input output relationship.

Anyone can assert a data point against the Data Asserter contract, with a known dataId and datapoint. Independent 3rd parties can then verify the correctness of this data against the dataId.

For our example we will be asserting the output of a very simple numerical computation:69+420 . This is meant to show the basic idea without overcomplicating the off-chain part. Note, though, that thedataId prop used here could represent any kind of unique data identifier (a row in a database table, a validator address or a complex structure used to associate any kind of data to a given value.

Development environment

This project usesforge as the Ethereum testing framework. You will also need to install Foundry, refer toFoundry installation documentation if you don’t have it already.

You will also needgit for cloning the repository, as well asbash shell andjq tool in order to parse transaction outputs when interacting with deployed contracts.

Clone the UMAOptimistic Oracle V3 quick-start repository and install the dependencies:

```

Copy gitclonehttps://github.com/UMAprotocol/dev-quickstart-oov3.git cddev-quickstart-oov3 forgeinstall

```

Contract implementation

The contract discussed in this tutorial can be found atdev-quickstart-oov3/src/DataAsserter.sol (here) within the repo.

Contract creation and initialization

To initialize the state variables of the contract, the constructor takes two parameters:

1. _defaultCurrency
2. parameter in the constructor identifies the token used for bonds of data assertions. This token should be approved as whitelisted UMA collateral. Please checkApproved Collateral Types
3. for production networks or callgetWhitelist()
4. on theAddress Whitelist
5. contract for any of the test networks. Note that the deployed Optimistic Oracle V3 instance already has itsdefaultCurrency
6. added to the whitelist, so it can also be used by the Data Asserter contract. Alternatively, you can approve a new token address withaddToWhitelist
7. method in the Address Whitelist contract if working in a sandboxed UMA environment.
8. _optimisticOracleV3
9. is used to locate the address of UMA Optimistic Oracle V3. Address ofOptimisticOracleV3
10. contact can be fetched from the relevantnetworks
11. file, if you are on a live network, or you can provide your own contract instance if deploying UMA Oracle contracts in your own sandboxed testing environment.
12.

```

Copy constructor(address_defaultCurrency,address_optimisticOracleV3) { defaultCurrency=IERC20(_defaultCurrency);
```

oo=OptimisticOracleV3Interface(_optimisticOracleV3); defaultIdentifier=oo.defaultIdentifier(); }

```
```

Asserting data

assertDataFor method allows any data provider to assert some value ofdata for the associateddataId . The caller sets theasserter as the address that will receive bonds at resolution of the data assertion (this could be their address as well).

```
```

Copy functionassertDataFor(bytes32dataId, bytes32data, addressasserter)publicreturns(bytes32assertionId) { ... }

```
```

Internally, this function pulls the associated assertion bond (global for all assertions within the DataAssertion contract) from the caller, makes an assertion call to the Optimistic Oracle V3 and stores the assertion within the contracts internalassertionsData mapping. Lastly, an event is emitted.

For the context of this example we will be considering thedataId simply as the URL that would be called to fetch the weather data fromwttr.in and thedata value as the result from this call. Both are bytes32 encoded.

The call to the Optimistic Oracle V3 looks as follows:

```
```

Copy assertionId=oo.assertTruth(abi.encodePacked("Data asserted: 0x",// in the example data is type bytes32 so we add the hex prefix 0x. ClaimData.toUtf8Bytes(data), " for dataId: 0x", ClaimData.toUtf8Bytes(dataId), " and asserter: 0x", ClaimData.toUtf8BytesAddress(asserter), " at timestamp: ", ClaimData.toUtf8BytesUint(block.timestamp), " in the DataAsserter contract at 0x", ClaimData.toUtf8BytesAddress(address(this)), " is valid."), asserter, address(this),// Callback recipient address(0),// No sovereign security. assertionLiveness, defaultCurrency, bond, defaultIdentifier, bytes32(0)// No domain.);

```
```

Lets break this down step by step:

- First, the value for theclaim
- filed is constructed to be a human readable string as the concatenation of a number of different data points. This is meant to be parsable by the validation network and enable both Humans and software verifiers alike to assess the correctness of the claim. Note that within the Optimistic Oracle V3 theclaim
- field (this prop in this function call) isnot
- stored on-chain; it is simply emitted and is used by off-chain actors to verify the correctness of the claim.
- Next, theasserter
- field is passed in from this function call. This is who will receive the bonds back at resolution of the assertion, assuming the asserter was correct.
- Next, we have the callback recipient set toaddress(this)
- . The OOV3 has the concept of callbacks wherein at the settlement of an assertion the asserter can choose to optionally call out from the OO into another contract. In this context, we want the OOV3 to callback
- into the Data Asserter contract when the assertion is settled (this will become clearer in the following section).
- We have no Sovereign security enabled here (this is an advanced topic and does not need to be covered here).
- Default liveness, bond and identifier are set.
- No domain is set (this is an external concept that helps 3rd parties identify assertions).
-

Resolving and subsequently fetching data assertions

Once data has been asserted, the act of verifying it is passed over to the Optimistic Oracle V3 and the associated network of data verifiers. Due to the presence of a bond (reward for catching invalid assertions), we can be confident that in real world someone would verify this assertion (if the bond is high enough, and the liveness is long enough we have high guarantees of this).

If the assertion passes theassertionLiveness without dispute then it can be settled. The data assertion contract showcases the OOV3's concept ofassertionResolvedCallback which enables the OOV3 to settle downstream contracts that are dependent on the resolution of an assertion. Once an assertion is settleable (it has: a) passed liveness and b) not been disputed) anyone can callsettleAssertion on the OOV3. This function acts to settle the assertion within the OO and call into the callback recipient, set during theassertDataFor function call.

Looking within the Data Asserter contract, we can see what happens when theassertionResolvedCallback is hit from the OOV3 on settlement:

```

Copy // OptimisticOracleV3 resolve callback. functionassertionResolvedCallback(bytes32assertionId,boolassertedTruthfully)public{ require(msg.sender==address(oo)); // If the assertion was true, then the data assertion is resolved. if(assertedTruthfully) { assertionsData[assertionId].resolved=true; DataAssertionmemorydataAssertion=assertionsData[assertionId]; emitDataAssertionResolved(dataAssertion.dataId,dataAssertion.data,dataAssertion.asserter,assertionId); // Else delete the data assertion if it was false to save gas. }elsedeleteassertionsData[assertionId]; }

```

This function effectively:

- enforce the inbound caller is the OOV3. Else, revert
- If the assertion was resolved as truthful (it would not be if it was disputed and the dispute proved to be correct) then:
-
    - set the data assertion status to true,
-
    - remove the data from the temporarydataAssertion
-
    - structure
- *
- if it was resolved as not truthful (deleted) then simply delete it from the temp structureassertionsData
-

Once settled, we can now call thegetData function with theassertionId. This will return the data value and the accompanying resolution status (was it resolved truthfully, or not).

Disputing data assertions

If we want to dispute an asserted data point one must simply call the[disputeAssertion(bytes32 assertionId, address disputer)] method on the OOV3. In here, you provide the associatedassertionId (the same field that was returned when callingassertData and thedisputer address (the address that will receive the bonds back at the resolution of the dispute).

Tests and deployment

All the unit tests covering the functionality described above are available[here] . To execute all of them, run:

```

Copy forgetest--match-path*DataAsserter*

```

Deployment

Before deploying and interacting with the contracts export the required environment variables. Note that there are a number of ways to interact with these contracts. The simplest setup is to fork Goerli testnet into a local development environment, using Anvil, and run the steps against the fork. To create an Anvil fork run the following:

```

Copy anvil--fork-urlhttps://goerli.infura.io/v3/xxx

```

Replacingxxx with your infura key. This will create a local fork of the Goerli testnet, running on127.0.0.1:8545 . It will also expose a test mnemonic that you can use which has pre-loaded Eth within the fork.

(Note: to set environment variables, as those listed below, use theexport syntax in your shell. for exampleexport ETH_RPC_URL=XXX

- ETHERSCAN_API_KEY
- : your secret API key used for contract verification on Etherscan if deploying on a public network (not required if you want to do this against a local testnet).
- ETH_RPC_URL
- : your RPC node used to interact with the selected network. For this tutorial it will be easiest to do this against the Goerli fork by settingETH_RPC_URL=http://127.0.0.1:8545
- MNEMONIC
- : your passphrase used to derive private keys of deployer (index 0) and any other addresses interacting with the contracts. The simplest setup with this is to re-use the Anvil default unlocked accounts, if you are using the local

testnet environment, you can have:

-
  ```
- Copy
- exportMNEMONIC="test test test test test test test test test test test junk"
- ```
- FINDER_ADDRESS
- : address of theFinder
- contract used to locate other UMA ecosystem contracts (in order to resolve disputes you would need to use the one from a sandboxed environment). For Goerli, you can use:
- ```
- Copy
- exportFINDER_ADDRESS=0xE60dBa66B85E10E7Fd18a67a6859E241A243950e
- ```
- 

Usecast command from Foundry to locate the address of Optimistic Oracle V3 and its default bonding token:

```

Copy exportOOV3_ADDRESS=(castcallFINDER_ADDRESS "getImplementationAddress(bytes32)(address)" \ (cast--format-bytes32-string"OptimisticOracleV3")) exportDEFAULT_CURRENCY_ADDRESS=(castcallOOV3_ADDRESS "defaultCurrency()(address)")

```

You should be able to see both of these values set in your env withecho OOV3_ADDRESS andecho DEFAULT_CURRENCY_ADDRESS

To deploy the Data Asserter contract, runforge create command.

(Note if you hit an error likeError accessing local wallet. Did you set a private key, mnemonic or keystore? then you might be hitting a foundry related issue that should be resolved by updating your foundry installation withfoundryup .

```

Copy exportDATA_ASSERTER=(forgecreate--jsonsrc/DataAsserter.sol:DataAsserter\ --mnemonic"MNEMONIC" \ --constructor-argsDEFAULT_CURRENCY_ADDRESS OOV3_ADDRESS \ |jq-r.deployedTo)

```

Finally, we can verify the deployed (if deployed to a public network) contract withforge verify-contract :

```

Copy forgeverify-contract\ --chain-id(castchain-id)\ --constructor-args(castabi-encode"constructor(address,address)" \ DEFAULT_CURRENCY_ADDRESS OOV3_ADDRESS)DATA_ASSERTERDataAsserter

```

Interacting with deployed contract

The following section provides instructions on how to interact with the deployed contract from the foundrycast tool, though one can also use it for guidance for interacting through another interface (e.g. Remix or Etherscan).

Initial setup

Export derivation index for the asserting user that will be needed when signing transactions:

```

Copy exportASSERTER_ID=1

```

Make sure the user address above have sufficient funding for the gas to execute the transactions.

Next, set up the bytes32 encoding for thedataId anddata , as discussed in theAsserting Data section. ThedataId field will be the simplest command that can be re-produced by independent verifiers. We usebc (arbitrary-precision arithmetic language and calculator) that should be available on most operating systems:

```

```
Copy exportDATA_ID=(cast--format-bytes32-string"echo '69+420' | bc") echo"DATA_ID=(echoDATA_ID)"
```

Thedata field will be the asserted output of the abovebc command:

```
Copy exportDATA=(cast--format-bytes32-string(echo'69+420'|bc)) echo"DATA=(echoDATA)"
```

Note that at any point you can look at your env by runningprintenv within your console to see the things we've set up to this point.

Asserting the data point

Next, we will call the Data Asserter contract and assert the data point and data ID that we set within our environment. Note that on the Goerli testnet the default bond for the default currency is set to 0. This is done to somewhat simplify the interactions: you don't need to mint yourself any USDC and you don't need to approve the Data Asserter to pull it from your account. Clearly, this is different to how this would be structured in reality but for demonstration purposes it keeps thing easier.

```
Copy exportASSERTION_TX=(castsend--json\ --mnemonic"MNEMONIC"--mnemonic-indexASSERTER_ID \ DATA_ASSERTER \ "assertDataFor(bytes32,bytes32,address)" DATA_ID DATA DATA_ASSERTER \ |jq-r.transactionHash)
```

What we are doing here is calling theassertDataFor(bytes32,bytes32,address) with the propsDATA_ID ,DATA andDATA_ASSERTER , that we set earlier. We are taking the output of the transaction and storing thetransactionHash within theASSERTION_TX variable.

We can fetch the associatedassertionId from theDataAssertionResolved event that should be the last event in the assertion transaction. The emitted event has the structure:

```
Copy eventDataAssertionResolved( bytes32indexeddataId, bytes32data, addressindexedasserter, bytes32indexedassertionId );
```

We can fetch indexed props from cast by using thereceipt function in conjunction withjq . To fetch the 3rd indexed prop (assertionId) from the last event, we would run:

```
Copy exportASSERTION_ID=(castreceipt--jsonASSERTION_TX|jq-r.logs[-1].topics[3])
```

Next, we can view the data asserted object using the--abi-decode call. This will return the props in theassertionData mapping, relating to this structure:

```
Copy structDataAssertion{ bytes32dataId;// The dataId that was asserted. bytes32data;// This could be an arbitrary data type. addressasserter;// The address that made the assertion. boolresolved;// Whether the assertion has been resolved. }
```

```
Copy cast--abi-decode"assertionsData(bytes32)(bytes32,bytes32,address,bool)"\ (castcallDATA_ASSERTER "assertionsData(bytes32)" ASSERTION_ID)
```

The last prop in the output (resolved) should befalse at this moment.

Settling the assertion

Now, let's move forward 2 hours to go pass the challenge window of the assertion:

```

Copy castrpcevm_increaseTime7200 castrpcevm_mine

```

Now, we can submit the settlement transaction to the OOV3 as we've passed liveness:

```

Copy castsend--mnemonic"MNEMONIC"OOV3_ADDRESS"settleAssertion(bytes32)"ASSERTION_ID

```

If we call theassertionData mapping again, we'll see that the assertion has settled. The last prop in the output (resolved) should betrue :

```

Copy cast--abi-decode"assertionsData(bytes32)(bytes32,bytes32,address,bool)"\ (castcallDATA_ASSERTER "assertionsData(bytes32)" ASSERTION_ID)

```

We can now call thegetData method directly and store output inIS_RESOLVED andRESOLVED_DATA variables:

```

Copy read-rIS_RESOLVEDRESOLVED_DATA\ <<(cast--abi-decode"getData(bytes32)(bool,bytes32)" \ (castcallDATA_ASSERTER "getData(bytes32)" ASSERTION_ID) \ |awk'{s=(NR==1?s:s " ")0}END{print s}')

```

IS_RESOLVED should now betrue :

```

Copy echoIS_RESOLVED

```

The decoded value ofRESOLVED_DATA should be489 as originally asserted:

```

Copy cast--parse-bytes32-stringRESOLVED_DATA

```

Conclusion

This tutorial has shown you how to use the Data Asserter sample tutorial, along with the Optimistic Oracle V3. This is meant to act as a starting point for further contracts to be built on. Next steps should be looking at the other OOV3 tutorials or trying to build your own integration! If you have questions please message us on Discord or create a Github issue and we'll happily help you with your integration.

Previous Insurance Next Escalation Managers Last updated1 month ago On this page *Data Asserter * Development environment * Contract implementation * Tests and deployment * Interacting with deployed contract * Conclusion

Was this helpful? Edit on GitHub