Insurance

Using the Optimistic Oracle V3 to allow for verification of insurance claims. This section covers the Insurance contract , which is available in the Optimistic Oracle V3quick-start repo . This tutorial shows an example of how insurance claims can be resolved and settled through theOptimistic Oracle V3 (OOV3) contract.

Insurance Contract

This smart contract allows insurers to issue insurance policies by depositing the insured amount, designating the insured beneficiary, and describing the insured event.

Anyone can request payout to the insured beneficiary at any time. The Insurance contract resolves the claim through the Optimistic Oracle V3 by asserting that the insured event has occurred as of request time using the default identifierASSERT_TRUTH specified inUMIP-170 .

If the claim is confirmed and settled through the Optimistic Oracle V3, this contract automatically pays out insurance coverage to the beneficiary. If the claim is rejected, the policy continues to be active and ready for subsequent claim attempts.

There is no limit to the number of payout requests that can be made of the same policy, however, only the first truthfully resolved request will settle the insurance payment, whereas the Optimistic Oracle V3 will settle bonds for all requests.

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/Insurance.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. identifies the token used for settlement of insurance claims, as well as the bond currency for assertions and disputes. 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 Insurance 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. contract 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(); }

```

Issuing insurance

issueInsurance method allows any insurer to depositinsuranceAmount ofdefaultCurrency tokens by designating an insurance beneficiary (payoutAddress) and defining the insured event (insuredEvent). Before calling this method, the insurer should have approved this contract to spend the required amount ofdefaultCurrency tokens.

```

Copy functionissueInsurance( uint256insuranceAmount, addresspayoutAddress, bytesmemoryinsuredEvent )publicreturns(bytes32policyId) { ... }

```

Internally, the issued policy is stored in thepolicies mapping using the calculatedpolicyId key that is generated by hashing the insured event and beneficiary address.

After pullinginsuranceAmount from the caller in theissueInsurance method, the contract emits aInsuranceIssued event including thepolicyId parameter that should be used when claiming insurance.

Submitting insurance claim

Anyone can submit an insurance claim on the issued policy by calling therequestPayout method with the relevantpolicyId parameter. This method will make an assertion with the Optimistic Oracle V3. An assertion bond is required, hence the caller should have approved this contract to spend the required minimum amount ofdefaultCurrency tokens for the proposal bond (callgetMinimumBond method on the Optimistic Oracle V3).

```

Copy functionrequestPayout(bytes32policyId)publicreturns(bytes32assertionId) { ... }

```

After checking that thepolicyId represents a valid insurance policy, the contract gets the currenttimestamp and composes claim that the insured event has occurred as of request time, that is passed to the Optimistic Oracle V3 when making the assertion with theassertTruth method:

```

Copy assertionId=oo.assertTruth( abi.encodePacked( "Insurance contract is claiming that insurance event ", policies[policyId].insuredEvent, " had occurred as of ", ClaimData.toUtf8BytesUint(block.timestamp), "." ), msg.sender,// asserter address(this),// callbackRecipient (the contract) address(0),// No sovereign security. assertionLiveness, defaultCurrency, bond, defaultIdentifier, bytes32(0)// No domain. ); assertedPolicies[assertionId]=policyId;

```

Optimistic Oracle V3 pulls the required bond and returnsassertionId that is used as a key when storing the linkedpolicyId in theassertedPolicies mapping. This information will be required when receiving a callback from the Optimistic Oracle V3.

Disputing insurance claim

For the sake of simplicity this contract does not implement a dispute method, but the disputer can dispute the submitted claim directly through Optimistic Oracle V3 before the liveness passes by calling itsdisputeAssertion method:

```

Copy functiondisputeAssertion(bytes32assertionId,addressdisputer)externalnonReentrant{ ... }

```

The disputer should pass theassertionId from the request above, as well as the address for receiving back bond and rewards if the disputer was right.

If the claim is disputed, the request is escalated to the UMA DVM and it can be settled only after UMA voters have resolved it. To learn more about the DVM, see the docs section on the DVM:how does UMA's Oracle work .

Settling insurance claim

Similar to disputes, claim settlement should be initiated through the Optimistic Oracle V3 contract by calling itssettleAssertion method with the sameassertionId parameter:

```
```

Copy functionsettleAssertion(bytes32assertionId)publicnonReentrant{ ... }

```

In case the liveness has expired or a dispute has been resolved by the UMA DVM, this call would initiate aassertionResolvedCallback callback in the Insurance contract:

```

Copy functionassertionResolvedCallback(bytes32assertionId,boolassertedTruthfully)public{ ... }

```

Importantly, all callbacks should be restricted to accept calls only from the Optimistic Oracle V3 to avoid someone spoofing a resolved answer:

```

Copy require(msg.sender==address(oo));

```

Depending on the resolved answer received in theassertedTruthfully callback parameter, this contract would either pay out the insured beneficiary if this was the first successful claim (in case oftrue representing that insurance claim was valid) or reject the payout:

```

Copy if(assertedTruthfully)_settlePayout(assertionId); ... function_settlePayout(bytes32assertionId)internal{ bytes32policyId=assertedPolicies[assertionId]; Policystoragepolicy=policies[policyId]; if(policy.settled)return; policy.settled=true; defaultCurrency.safeTransfer(policy.payoutAddress,policy.insuranceAmount); emitInsurancePayoutSettled(policyId,assertionId); }

```

Tests and deployment

All the unit tests covering the functionality described above are availablehere . To execute all of them, run:

```

Copy forgetest--match-path*Insurance*

```

Deployment

Before deploying and interacting with the contracts export the required environment variables:

- ETHERSCAN_API_KEY
- : your secret API key used for contract verification on Etherscan if deploying on a public network
- ETH_RPC_URL
- : your RPC node used to interact with the selected network
- MNEMONIC
- : your passphrase used to derive private keys of deployer (index 0) and any other addresses interacting with the contracts
- 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
- ```
- DEFAULT_CURRENCY_ADDRESS
- : address of the token used for insurance claim settlement. This is also used as oracle bonding currency, thus, needs to be added to whitelist either by UMA governance (production networks) or testnet administrator. On Goerli you can use0xe9448D94C9b033Ff50d3B14089043bD976fC1394
- that is already whitelisted and can be minted by anyone using itsallocateTo
- method
-
```

Usecast command from Foundry to locate the address of Optimistic Oracle V3:

```

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

```

To deploy the Insurance contract, runforge create command.

```

Copy exportINSURANCE_ADDRESS=(forgecreate--jsonsrc/Insurance.sol:Insurance\ --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)\ INSURANCE_ADDRESSInsurance

```

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 required user addresses and their derivation indices:

```

Copy exportINSURER_ID=1 exportINSURED_ID=2 exportINSURER_ADDRESS=(castwalletaddress--mnemonic"MNEMONIC"--mnemonic-indexINSURER_ID) exportINSURED_ADDRESS=(castwalletaddress--mnemonic"MNEMONIC"--mnemonic-indexINSURED_ID)

```

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

Issue insurance

Make sure to have some amount ofDEFAULT_CURRENCY_ADDRESS tokens to back potential insurance claim. If0xe9448D94C9b033Ff50d3B14089043bD976fC1394 was used on Goerli you can mint 10,000 DBT tokens to insurance issuer account:

```

Copy exportINSURANCE_AMOUNT=(cast--to-wei10000) castsend--mnemonic"MNEMONIC"--mnemonic-indexINSURER_ID \ DEFAULT_CURRENCY_ADDRESS"allocateTo(address,uint256)"INSURER_ADDRESS INSURANCE_AMOUNT

```

ApproveDEFAULT_CURRENCY_ADDRESS to be pulled by the Insurance contract:

```

Copy castsend--mnemonic"MNEMONIC"--mnemonic-indexINSURER_ID \ DEFAULT_CURRENCY_ADDRESS"approve(address,uint256)"INSURANCE_ADDRESS INSURANCE_AMOUNT

```

Issue the insurance policy and grab the resultingpolicyId from the emittedInsuranceIssued event (it should be the last emitted event in the transaction and indexedpolicyId is at topic index1):

```
```

Copy exportISSUE_TX=(castsend--json\ --mnemonic"MNEMONIC"--mnemonic-indexINSURER_ID \
INSURANCE_ADDRESS \ "issueInsurance(uint256,address,bytes)" \ INSURANCE_AMOUNT INSURED_ADDRESS (cast--
from-utf8"Bad things have happened") \ |jq-r.transactionHash) exportPOLICY_ID=(castreceipt--jsonISSUE_TX|jq-r.logs[-
1].topics[1])

```

If in doubt on the parsing of transaction receipt you can manually view full logs from tracing:

```

Copy castrunISSUE_TX

```

Submit insurance claim

Get the expected oracle bond:

```

Copy exportBOND_AMOUNT=(castcallOOV3_ADDRESS \ "getMinimumBond(address)(uint256)"
DEFAULT_CURRENCY_ADDRESS)

```

This should be zero for0xe9448D94C9b033Ff50d3B14089043bD976fC1394 on Goerli, but in case of other currencies make sure to have this amount ofDEFAULT_CURRENCY_ADDRESS both on the insured account (for submitting the claim) and on the insurer's account (for disputing the claim). If bond amount is non-zero, also make sure to add approval:

```

Copy castsend--mnemonic"MNEMONIC"--mnemonic-indexINSURED_ID \
DEFAULT_CURRENCY_ADDRESS"approve(address,uint256)"INSURANCE_ADDRESS BOND_AMOUNT

```

Now initiate the insurance claim and grab the resultingassertionId from the emittedInsurancePayoutRequested event (it should be the last emitted event and indexedassertionId is at topic index2 ):

```

Copy exportASSERTION_TX=(castsend--json\ --mnemonic"MNEMONIC"--mnemonic-indexINSURED_ID \
INSURANCE_ADDRESS \ "requestPayout(bytes32)" POLICY_ID|jq-r.transactionHash) exportASSERTION_ID=(castreceipt-
-jsonASSERTION_TX|jq-r.logs[-1].topics[2])

```

If in doubt on the parsing of transaction receipt you can manually view full logs from tracing:

```

Copy castrunASSERTION_TX

```

Dispute insurance claim

Before liveness passes, anyone (e.g. insurer) can dispute the claim through the Optimistic Oracle V3. In case of non-zero bond amount, they must add approval for Optimistic Oracle V3 to pull the bond:

```

Copy castsend--mnemonic"MNEMONIC"--mnemonic-indexINSURER_ID \
DEFAULT_CURRENCY_ADDRESS"approve(address,uint256)"OOV3_ADDRESS BOND_AMOUNT

```

Now initiate the dispute and export related transaction hash that we will need to collect additional request parameters for resolving the dispute:

```

Copy exportDISPUTE_TX=(castsend--json\ --mnemonic"MNEMONIC"--mnemonic-indexINSURER_ID \ OOV3_ADDRESS "disputeAssertion(bytes32,address)" ASSERTION_ID INSURER_ADDRESS \ |jq-r.transactionHash)

```

Settle insurance claim

Resolving disputes in production environment involves UMA token holders to vote on the request. Thus, testing is possible only in a sandboxed UMA ecosystem environment where theMock Oracle is used to resolve requests:

```

Copy exportMOCK_ORACLE_ADDRESS=(castcall\ FINDER_ADDRESS "getImplementationAddress(bytes32)(address)" \ (cast--format-bytes32-string"Oracle"))

```

In order to resolve request Mock Oracle expects the following parameters:

- identifier
- : identifier that references instructions to resolve the request (Optimistic Oracle V3 and Insurance contract uses default identifierASSERT_TRUTH
- specified inUMIP-170
- .)
- time
- : timestamp when the disputed assertion was made
- ancillaryData
- : bytes encoded additional data required to resolve the request (Optimistic Oracle V3 includesassertionId
- and the address of the claiming asserter)
- price
- : numerical value to decide the outcome of the request (Optimistic Oracle V3 requires this to be1e18
- in order to resolve assertion as truthful)
-

The commands below export the required parameters and resolves the request at the Mock Oracle:

```

Copy exportIDENTIFIER=(castcallINSURANCE_ADDRESS "defaultIdentifier()(bytes32)") exportASSERTION_TIME= (castblock--json\ (casttx--jsonASSERTION_TX|jq-r.blockNumber)|jq-r.timestamp) exportANCILLARY_DATA=(cast--from-utf8(echoassertionId:(echoASSERTION_ID|sed's/0x//'|tr[:upper:] [:lower:]),ooAsserter: (echoINSURED_ADDRESS|sed's/0x//'|tr[:upper:] [:lower:])))

exportPRICE=(cast--to-wei1) castsend--mnemonic"MNEMONIC"--mnemonic-indexINSURED_ID \ MOCK_ORACLE_ADDRESS \ "pushPrice(bytes32,uint256,bytes,int256)"\ IDENTIFIER ASSERTION_TIME ANCILLARY_DATA PRICE

```

Note that the syntax above for getting ancillary data involves stripping0x and transformingassertionId andooAsserter to lowercase to replicate the logic how Optimistic Oracle V3 contract is composing the ancillary data. If in doubt, you can always check the emittedPriceRequestAdded event parameters from the Mock Oracle contract in the dispute transaction traces:

```

Copy castrunDISPUTE_TX

```

Now we can settle the request through the Optimistic Oracle V3 and observe the emittedInsurancePayoutSettled from our Insurance contract:

```

Copy exportSETTLE_TX=(castsend--json\ --mnemonic"MNEMONIC"--mnemonic-indexINSURED_ID \ OOV3_ADDRESS "settleAssertion(bytes32)" ASSERTION_ID \ |jq-r.transactionHash) castrunSETTLE_TX

```

The above settlement transaction should also transferINSURANCE_AMOUNT tokens to the insured beneficiary as well as return the assertion bond plus half of disputer's bond to the claim initiator.
```

Alternatively, if0 value was resolved inPRICE (representing the asserted claim was not true), the settlement transaction should only return the bond plus half of claim initiator's bond to the disputer.

Previous Prediction Market Next Data Asserter Last updated2 months ago On this page *Insurance Contract * Development environment * Contract implementation * Tests and deployment * Interacting with deployed contract