Prediction Market

Building a prediction market that uses the UMA Optimistic Oracle V3 for settlement and event identification This section covers thePrediction Market contract , which is available in the Optimistic Oracle V3quick-start repo and enables the creation of a binary prediction market usingOptimistic Oracle V3 (OOV3) assertions.

Prediction Market

A prediction market smart contract allows individuals to make predictions and place bets on the likelihood of different outcomes for a future event. These types of contracts can be used for any kind of events such as:

1. Sports games
2. Cryptocurrency price predictions
3. Product launches
4. Political policy decisions
5.

This smart contract enables the creation of prediction markets based on any off-chain event with three possible outcomes, two of which are chosen by the market creator and the third reflecting the remaining outcomes (i.e. tie or a draw in a sporting event).

By participating in the prediction market, individuals can create outcome tokens (shares) and buy and sell them of the different outcomes based on their assessment of the likelihood of each outcome occurring. The market determines the price of each share based on supply and demand, and individuals can profit by purchasing shares whose value grows as the likelihood of a certain event increases. Nevertheless, this contract does not cover the selling of outcome tokens; only their creation is covered.

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/PredictionMarket.sol (here) within the repo.

Contract creation and initialization

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

1. _finder
2. : address that represents the Finder contract, which is used to locate and retrieve other contracts within the system.
3. _currency
4. : address that represents the currency that will be used for trading in the markets created by this contract.
5. _optimisticOracleV3
6. : address that represents the OptimisticOracleV3, contract to assert truths about the world which are verified using an optimistic escalation game.
7.

```

Copy constructor( address_finder, address_currency, address_optimisticOracleV3 ) { finder=FinderInterface(_finder); require(_getCollateralWhitelist().isOnWhitelist(_currency),"Unsupported currency"); currency=IERC20(_currency); oo=OptimisticOracleV3Interface(_optimisticOracleV3); defaultIdentifier=oo.defaultIdentifier(); }

```

Market creation

TheinitializeMarket() function is used to create a new market in the Prediction Market contract.

Once the contract has been deployed, anyone can callinitializeMarket() after approving the optionalreward amount to be paid to the wallet that runs the assertion.

```

Copy functioninitializeMarket( stringmemoryoutcome1,// Short name of the first outcome. stringmemoryoutcome2,// Short name of the second outcome. stringmemorydescription,// Description of the market. uint256reward,// Reward available for asserting true market outcome. uint256requiredBond// Expected bond to assert market outcome (OOv3 can require higher bond). )publicreturns(bytes32marketId) { ... }

```

To create a new market, theinitializeMarket() function is called with the following parameters:

- outcome1
- : A short name for the first outcome (i.e., "yes").
- outcome2
- : A short name for the second outcome (i.e., "no").
- description
- : A description of the market and the event being predicted.
- reward
- : The amount of currency (in Wei) that will be available as a reward for the user that runs that creates the assertion in the OOV3, necessary to settle the market once it's ready.
- requiredBond
- : The amount of currency (in Wei) that users must bond to assert the outcome of the event.
-

Once the input checks are passed, the function creates two new ERC20 tokens (one for each outcome) using theExpandedERC20 contract, which extends the standard ERC20 contract by adding the ability to mint and burn tokens. The Prediction Market is assigned the minter and burner roles to simplify how the outcome tokens handled when creating, redeeming or settling the tokens.

TheMarket is then created and stored internally associated to amarketId returned by theinitializeMarket() and emitted in theMarketInitialized event at the end of the process together with the parameters defining the market.

Create outcome tokens

ThecreateOutcomeTokens function mints a pair of tokens representing the value of outcome1 and outcome2 for a given market identified bymarketId . This allows participants to trade on the outcome of the market by exchanging these tokens with each other outside of the scope of the Prediction Market contract.

ThecreateOutcomeTokens function takes two parameters:

- tokensToCreate
- : an unsigned integer (uint256
-) value representing the number of tokens to be created for each outcome. The total number of tokens created will be twice this value, as two outcome tokens are created for each market.
- marketId
- : abytes32
- value representing the unique identifier of the market for which the outcome tokens are being created.
-

The function first checks if the market exists. If the market exists, it transfers the specified amount of currency tokens from the caller to the contract usingsafeTransferFrom function fromcurrency contract instance.

Then, the function mintstokensToCreate amount of both outcome1 and outcome2 tokens to the caller using themint function of the respectiveExpandedERC20 token instances. Finally, it emits an event to notify that tokens have been created.

It is important to note that before calling this function, the caller must approve the contract to spend the required amount of currency tokens by calling theapprove function on thecurrency contract instance.

Redeem outcome tokens

TheredeemOutcomeTokens function burns an equal amount ofoutcome1Token andoutcome2Token tokens for a given market and transfers the corresponding settlement currency tokens to the caller's account.

The function takes two parameters:

- marketId

- : abytes32
- value representing the unique identifier of the market for which the tokens are being redeemed.
- tokensToRedeem
- : an unsigned integer (uint256
-) value representing the number of tokens of each outcome to be redeemed. The total number of tokens redeemed will be twice this value, as two outcome tokens are burned for each market.
-

Both parameters are passed to the function as arguments when it is called.

The function first retrieves the Market data from the storage using the provided marketId. Then, it checks if the outcome1Token is not equal to the zero address, which indicates the market exists.

Next, it burns the specified number of tokens for both outcome1Token and outcome2Token tokens using theburnFrom() function of their respective contracts, which decreases the balance of the caller's account.

Finally, it transfers the specified number of settlement currency tokens from the contract's account to the caller's account using thesafeTransfer() function. The function emits aTokensRedeemed event, which includes the marketId, the caller's address, and the number of tokens redeemed.

Settle outcome tokens

ThesettleOutcomeTokens function allows a user to settle their outcome tokens for a given market and receive a payout in the settlement currency. The payout depends on the resolved market outcome and the number of tokens burned for each outcome.

The function takes one parameter:

- marketId
- : abytes32
- value representing the unique identifier of the market for which the outcome tokens are being settled.
-

ThemarketId parameter is passed to the function as an argument when it is called.

The function first retrieves the Market data from the storage using the providedmarketId . It then checks if the market has been resolved by verifying that theresolved flag in the market struct is set totrue . If the market has not been resolved, the function throws an error.

Next, the function determines how manyoutcome1Token andoutcome2Token tokens the caller has by calling thebalanceOf() function of their respective contracts. The balances are stored inoutcome1Balance andoutcome2Balance variables.

After that, the function calculates the payout based on the resolved market outcome and the number of tokens burned for each outcome. If the market was resolved to the first outcome, then the payout equals the balance ofoutcome1Token whileoutcome2Token provides nothing. If the market was resolved to the second outcome, then the payout equals the balance ofoutcome2Token whileoutcome1Token provides nothing. If the market was resolved to the split outcome, then both outcome tokens provide half of their balance as currency payout.

Finally, the function burns the caller's outcome tokens using theburnFrom() function of their respective contracts, transfers the calculated payout from the contract's account to the caller's account using thesafeTransfer() function, and emits aTokensSettled event, which includes the marketId, the caller's address, the payout amount, and the number ofoutcome1Token andoutcome2Token tokens burned. The calculated payout is returned by the function.

Assert market

The functionassertMarket is used to assert a market with any of the three possible outcomes: outcome1, outcome2, or unresolvable. The function takes two arguments:

- marketId
- , which is a unique identifier for the market
- assertedOutcome
- , which is a string representing the asserted outcome.
-

The function first checks that the market exists by verifying that the outcome1 token is not a null address. It then computes the hash of the asserted outcome using thekeccak256 function and checks that the market does not already have an active or resolved assertion. It also checks that the asserted outcome is one of the three allowed outcomes.

If all checks pass, the function sets theassertedOutcomeId of the market to the hash of the asserted outcome and computes the bond required to make the assertion. If the bond required by the market is higher than the minimum bond required by the oracle, the market bond is used instead. The function then composes a claim with the asserted outcome and the market

description, pulls the bond from the caller's account, approves it for the oracle, and makes the assertion using the_assertTruthWithDefaults function.

Finally, the function stores information about the asserted market and emits aMarketAsserted event with the market ID, the asserted outcome, and the assertion ID.

Tests and deployment

To run the contracts tests written in solidity with forge run the following command:

```

Copy forgetest--match-path*PredictionMarket*

```

Deployment

Run a local node with anvil (included in foundry toolkit) in a console:

```

Copy anvil

```

In a second console continue running the rest of commands. Let's first export the environment variables for the wallets to use:

```

Copy exportMNEMONIC="test test test test test test test test test test test junk" exportUSER_ID=1 exportASSERTER_ID=2 exportDEPLOYER_WALLET=(castwalletaddress--mnemonic"MNEMONIC") exportUSER_WALLET=(castwalletaddress--mnemonic"MNEMONIC"--mnemonic-indexUSER_ID) exportASSERTER_WALLET=(castwalletaddress--mnemonic"MNEMONIC"--mnemonic-indexASSERTER_ID) exportETH_RPC_URL="http://127.0.0.1:8545"

```

Then Deploy the UMA Oracle Sandbox contracts to be used by running:

```

Copy forgescriptscript/OracleSandbox.s.sol\ --fork-urlETH_RPC_URL \ --mnemonics"MNEMONIC"\ --senderDEPLOYER_WALLET \ --broadcast

```

Find in the logs theFINDER_ADDRESS and export it with:

```

Copy exportFINDER_ADDRESS=

```

Then use the Finder to export rest of addresses by running the following commands:

```

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

```

We are ready to deploy the Prediction Market contract with the following command:

```

Copy exportPREDICTION_MARKET_ADDRESS=(forgecreatesrc/PredictionMarket.sol:PredictionMarket\ --json\ --mnemonic"MNEMONIC" \ --constructor-argsFINDER_ADDRESS DEFAULT_CURRENCY_ADDRESS OOV3_ADDRESS \ |jq-r.deployedTo) echo"PREDICTION MARKET DEPLOYED TO"PREDICTION_MARKET_ADDRESS
```

```

Interacting with deployed contract

It's time to initialise a market. We can first export the market parameters to use.

Here the description shows that it's a market based on the outcome of a match between two teams. The two possible outcomes areyes orno. We offer 100 units ofDEFAULT_CURRENCY to the asserter of the claim and require a bond of 5,000 units ofDEFAULT_CURRENCY to assert or dispute the assertion.

```

Copy exportDESCRIPTION="The Glacial Storms beat the Electric Titans on March 8, 2023 at 3:00 PM UTC, \ which is equivalent to the Unix timestamp 1686258000 seconds." exportOUTCOME_ONE="yes" exportOUTCOME_TWO="no" exportREWARD=(cast--to-wei100) exportREQUIRED_BOND=(cast--to-wei5000)

```

We need to mint the amount of asserter rewards and approve them before creating the market:

```

Copy castsend--mnemonic"MNEMONIC"\
DEFAULT_CURRENCY_ADDRESS"allocateTo(address,uint256)"DEPLOYER_WALLET REWARD castsend--mnemonic"MNEMONIC"\
DEFAULT_CURRENCY_ADDRESS"approve(address,uint256)"PREDICTION_MARKET_ADDRESS REWARD

```

Then we are ready to initialise the market with theDEPLOYER_WALLET

```

Copy exportMARKET_ID_TX=(castsend--json\ --mnemonic"MNEMONIC" \ PREDICTION_MARKET_ADDRESS \ "initializeMarket(string,string,string,uint256,uint256)(bytes32)" \ OUTCOME_ONE OUTCOME_TWO "DESCRIPTION" REWARD REQUIRED_BOND \ |jq-r.transactionHash) exportMARKET_ID=(castreceipt--jsonMARKET_ID_TX|jq-r.logs[-1].topics[1]) exportOUTCOME_TOKEN_ONE_ADDRESS=(cast--abi-decode\ "MarketInitializedNonIndexed()(string,string,string,address,address,uint256,uint256)" \ (castreceipt--jsonMARKET_ID_TX|jq-r.logs[-1].data)|awk'NR==4{print 1}') exportOUTCOME_TOKEN_TWO_ADDRESS=(cast--abi-decode\ "MarketInitializedNonIndexed()(string,string,string,address,address,uint256,uint256)" \ (castreceipt--jsonMARKET_ID_TX|jq-r.logs[-1].data)|awk'NR==5{print 1}')

```

Then we can mint the necessary tokens to then create the outcome tokens:

```

Copy exportAMOUNT=(cast--to-wei10000) castsend--mnemonic"MNEMONIC"\
DEFAULT_CURRENCY_ADDRESS"allocateTo(address,uint256)"DEPLOYER_WALLET AMOUNT castsend--mnemonic"MNEMONIC"\
DEFAULT_CURRENCY_ADDRESS"approve(address,uint256)"PREDICTION_MARKET_ADDRESS AMOUNT

```

We can now create the outcome tokens. With an amount 10,000 units ofDEFAULT_CURRENCY we get 10,000OUTCOME_TOKEN_ONE and 10,000OUTCOME_TOKEN_TWO tokens:

```

Copy castsend--mnemonic"MNEMONIC"\
PREDICTION_MARKET_ADDRESS"createOutcomeTokens(bytes32,uint256)"MARKET_ID AMOUNT

```

```

Copy echo"BALANCE OUTCOME TOKEN ONE"(castcallOUTCOME_TOKEN_ONE_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"BALANCE OUTCOME TOKEN TWO"(castcallOUTCOME_TOKEN_TWO_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"BALANCE DEFAULT_CURRENCY"(castcallDEFAULT_CURRENCY_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET)
```
```

```

At any point before the market is settled we can redeem outcome tokens. By redeeming an amount we are burning the same amount ofOUTCOME_TOKEN_ONE andOUTCOME_TOKEN_TWO to receive that amount ofDEFAULT_CURRENCY :

```

Copy castsend--mnemonic"MNEMONIC"\ PREDICTION_MARKET_ADDRESS"redeemOutcomeTokens(bytes32,uint256)"\ MARKET_ID(cast--to-wei5000)

```

After redeeming 5,000 tokens we can see how both balances ofOUTCOME_TOKEN_ONE andOUTCOME_TOKEN_TWO have decreased by 5,000 andDEFAULT_CURRENCY has increased that same amount.

```

Copy echo"BALANCE OUTCOME TOKEN ONE"(castcallOUTCOME_TOKEN_ONE_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"BALANCE OUTCOME TOKEN TWO" (castcallOUTCOME_TOKEN_TWO_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"BALANCE DEFAULT_CURRENCY"(castcallDEFAULT_CURRENCY_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET)

```

Now, let's simulate how theDEPLOYER_WALLET would trade one position of the market by transferring the remaining 5,000OUTCOME_TOKEN_ONE to another user. By doing this,DEPLOYER_WALLET is now only exposed to the outcome two ("no") because he only holdsOUTCOME_TOKEN_TWO . On the other side,USER_WALLET is exposed to the outcome one ("yes") as he has traded some other currency againstOUTCOME_TOKEN_ONE . This trade is out of the scope of this example, thats why we simulate it by running the following transfer:

```

Copy castsend--mnemonic"MNEMONIC"\ OUTCOME_TOKEN_ONE_ADDRESS"transfer(address,uint256)"USER_WALLET(cast--to-wei5000)

```

At this point, let's imagine that the match between The Glacial Storms and the Electric Titans has taken place and that The Glacial Storms won. Then anyone can nowassert that this has occurred by callingassertMarket with outcome"yes" as the claim defined inDESCRIPTION is true. We can do it, from theASSERTER_WALLET , by running the following command:

```

Copy castsend--mnemonic"MNEMONIC"\ DEFAULT_CURRENCY_ADDRESS"allocateTo(address,uint256)"\ ASSERTER_WALLET REQUIRED_BOND castsend--mnemonic"MNEMONIC"--mnemonic-indexASSERTER_ID \ DEFAULT_CURRENCY_ADDRESS"approve(address,uint256)"\ PREDICTION_MARKET_ADDRESS REQUIRED_BOND exportASSERTION_TX=(castsend--json\ --mnemonic"MNEMONIC"--mnemonic-indexASSERTER_ID \ PREDICTION_MARKET_ADDRESS "assertMarket(bytes32,string)" MARKET_ID "yes" \ |jq-r.transactionHash) exportASSERTION_ID=(castreceipt--jsonASSERTION_TX|jq-r.logs[-1].topics[2])

```

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

```

Copy castrpcevm_increaseTime7200 castrpcevm_mine

```

Now the assertion can be settled in theOptimisticOracleV3 . We can do it by running the following command:

```

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

```

We can now check how theASSERTER_WALLET has received back the assertion bond plus the reward:

```

Copy echo"ASSERTER WALLET BALANCE DEFAULT_CURRENCY"\ (castcallDEFAULT_CURRENCY_ADDRESS "balanceOf(address)(uint256)" ASSERTER_WALLET)

```

Now, both theDEPLOYER_WALLET andUSER_WALLET can settle their outcome tokens:

```

Copy castsend--mnemonic"MNEMONIC"\ PREDICTION_MARKET_ADDRESS"settleOutcomeTokens(bytes32)"MARKET_ID castsend--mnemonic"MNEMONIC"--mnemonic-indexUSER_ID \ PREDICTION_MARKET_ADDRESS"settleOutcomeTokens(bytes32)"MARKET_ID

```

Finally we can see how theUSER_WALLET won the bet, as he gotOUTCOME_TOKEN_ONE so he now has 5,000DEFAULT_CURRENCY and the deployer wallet only has 5,000DEFAULT_CURRENCY from his initial 10,000:

```

Copy echo"DEPLOYER WALLET BALANCE OUTCOME TOKEN ONE"(castcallOUTCOME_TOKEN_ONE_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"DEPLOYER WALLET BALANCE OUTCOME TOKEN TWO" (castcallOUTCOME_TOKEN_TWO_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"DEPLOYER WALLET BALANCE DEFAULT_CURRENCY"(castcallDEFAULT_CURRENCY_ADDRESS \ "balanceOf(address)(uint256)" DEPLOYER_WALLET) echo"USER WALLET BALANCE OUTCOME TOKEN ONE" (castcallOUTCOME_TOKEN_ONE_ADDRESS \ "balanceOf(address)(uint256)" USER_WALLET) echo"USER WALLET BALANCE OUTCOME TOKEN TWO"(castcallOUTCOME_TOKEN_TWO_ADDRESS \ "balanceOf(address)(uint256)" USER_WALLET) echo"USER WALLET BALANCE DEFAULT_CURRENCY"(castcallDEFAULT_CURRENCY_ADDRESS \ "balanceOf(address)(uint256)" USER_WALLET)

```

Previous Optimistic Oracle Next Insurance Last updated1 month ago On this page *Prediction Market * Development environment * Contract implementation * Tests and deployment * Interacting with deployed contract
```