ComposableCoW

ComposableCoW is a framework for smoothing the developer experience when building conditional orders on CoW Protocol. Conditional orders are a subset ofERC-1271 smart contract orders. It allows one to create conditional orders that:

1. Can be used to generate multiple discrete order (self-expressing)
2. Assess a proposed order against a set of conditions (self-validating)

The framework makes boilerplate code for conditional orders a thing of the past, and allows developers to focus on the business logic of their order. ComposableCoW handles:

1. Authorization (multiple owners, with multiple orders per owner)
2. Order relaying (watch-towers)

Architecture

The following principles have been employed in the architectural design:

1. O(1)
2. gas-efficiency forn
3. conditional order creation / replacement / deletion
4. Conditional ordersSHOULD
5. behave the same as a discrete order for EOAs (self-custody of assets, i.e. "wrapper" contracts not required)
6. Conditional ordersSHOULD
7. be optimized towardsstatelessness
8.
    - pass required data viacalldata
9. MAY
10. enhance theSafe
11. user experience when paired withExtensibleFallbackHandler
12. 🐮

By using Merkle Trees, the gas efficiency ofO(1) is achieved forn conditional orders. This is achieved by storing the Merkle Tree root on-chain, and passing the Merkle Tree proof to theComposableCoW contract. This allows forO(1) gas efficiency for adding / removing conditional orders.

For simplicity, single orders are also supported, however, this isNOT recommended for largen as the gas efficiency isO(n) .

Execution context

As there are many nested contracts, it's important for a callee to know some context from the caller. To achieve this, ComposableCoW passes abytes32 variablectx to the callee, such that:

ctx = merkle root of orders: bytes32(0) single order: H(ConditionalOrderParams) Having this context also allows for conditional orders / merkle roots to use this as a key in a mapping, to store conditional order-specific data.

Conditional order verification flow

The following flowchart illustrates the conditional order verification flow (assumingsafe):

Settlement execution path

CoW Protocol order settlement execution path (assumingsafe):

Signature verification

ComposableCoW implementsISafeSignatureVerifier , which allows for delegatedERC-1271 signature validation with an enhanced context:

function

isValidSafeSignature (Safe safe , address sender , bytes32 _hash , bytes32 domainSeparator , bytes32 ,

// typeHash bytes

calldata encodeData , bytes

calldata payload)

external

view override returns

(bytes4 magic) ; Parameter Description safe Contract that is delegating signing sender msg.sender that calledisValidSignature onsafe _hash Order digest domainSeparator SeeEIP-712 typeHash Not used encodeData ABI-encodedGPv2Order.Data (perEIP-712) to be settled encodeData ABI-encodedGPv2Order.Data to be settled In order to delegate signature verification toComposableCoW , the delegating contract may either:

1. Be a Safe and useExtensibleFallbackHandler
2. that allows forEIP-712
3. domain delegation to a custom contract (i.e.ComposableCoW
4.); or
5. ImplementERC-1271
6. and within theisValidSignature
7. method, callComposableCoW.isValidSafeSignature()
8. .

tip ComposableCoW can also be used with contracts other than Safe. TheERC1271Forwarder abstract contract has been provided to allow for new contracts to easily integrate with ComposableCoW. note If usingExtensibleFallbackHandler , and the CoW Protocol settlement domain is delegated toComposableCoW ,ALL ERC-1271 signatures will be processed byComposableCoW .

Discrete order verifiers

A conditional order that verifies a proposed discrete order against a set of conditions shall implement theIConditionalOrder interface.

function

verify (address owner , address sender , bytes32 _hash , bytes32 domainSeparator , bytes32 ctx , bytes

calldata staticInput , bytes

calldata offchainInput GPv2Order . Data calldata order ,)

external

view ; Parameter Description owner The owner of the conditional order sender msg.sender context callingisValidSignature _hash EIP-712 order digest domainSeparator EIP-712 domain separator ctx Execution context staticInput Conditional order type-specific data known at time of creation forall discrete orders offchainInput Conditional order type-specific dataNOT known at time of creation for aspecific discrete order (or zero-length bytes if not applicable) order The proposed discrete order'sGPv2Order.Data struct warning Order implementationsMUST validate / verifyoffchainInput ! caution Theverify methodMUST revert withOrderNotValid(string) if the parameters instaticInput do not correspond to a valid order. note ComposableCoW is responsible for checking that all valuesEXCLUDING offchainInput belong to an order that was previously registered on-chain.

Discrete order generators

A conditional order that generates discrete orders shall implement theIConditionalOrderGenerator interface.

function

getTradeableOrder (address owner , address sender , bytes32 ctx , bytes

calldata staticInput , bytes

calldata offchainInput)

external

view

returns

(GPv2Order . Data memory) ; To simplify the developer experience, aBaseConditionalOrder contract has been provided that implements theIConditionalOrderGenerator interface, and necessary boilerplate.

Swap guards

A swap guard is a contract that implements theISwapGuard interface, and if set by anowner , will be called byComposableCoW prior to callingverify on the conditional order.

This allows forowner -wide restrictions on the conditional order, such as:

- receiver lock
- (i.e.receiver
- MUST
- beowner
-)
- Token whitelist

TheISwapGuard interface is as follows:

function

verify (GPv2Order . Data calldata order , bytes32 ctx , IConditionalOrder . ConditionalOrderParams calldata params , bytes

calldata offchainInput)

external

view

returns

(bool) ; Parameter Description order Proposed discrete order ctxExecution context params ConditionalOrderParams offchainInput Conditional order type-specific dataNOT known at time of creation for aspecific discrete order (or zero-length bytes if not applicable)

Guarantees and Invariants

- CoW Protocol's settlement contract enforces single-use orders, i.e.NO
- GPv2Order
- can be filled more than once
- For merkle trees,H(ConditionalOrderParams)
- MUST
- be a member of the merkle treeroots[owner]
- For single orders,singleOrders[owner][H(ConditionalOrderParams)] == true

caution While a discrete order can be filled only once on CoW Protocol, a single conditional order can be used to create many different discrete orders. It is the responsibility of the implementation to limit which and when discrete orders can be executed.

Data Types and Storage

ConditionalOrderParams

A conditional order is defined by the following data:

struct

ConditionalOrderParams

{ IConditionalOrder handler ; bytes32 salt ; bytes staticData ; } Field Description handler The contract implementing the conditional order logic salt Allows for multiple conditional orders of the same type and data staticData Data available toALL discrete orders created by the conditional order note All of the above fields are verified byComposableCoW to be valid, prior to calling theverify method on the handler (IConditionalOrder). tip When used with Merkle Trees and a cryptographically-secure randomsalt , the conditional order is effectively private (until a discrete order cut from this conditional order is broadcast to the CoW Protocol API). caution * H(ConditionalOrderParams) * MUST * be unique * Not settingsalt * to a cryptographically-secure random valueMAY * result in leaking information or hash collisions * Single ordersMAY * leak order information on creation

PayloadStruct

This is the data passed toComposableCoW via thepayload parameter ofisValidSafeSignature :

struct

PayloadStruct

{ bytes32 [ ] proof ; IConditionalOrder . ConditionalOrderParams params ; bytes offchainInput ; } Field Description proof Merkle Tree proof (if applicable, zero length otherwise) params ConditionalOrderParams offchainInput Off-chain input (if applicable, zero length otherwise) By settingproof to zero-length, this indicates toComposableCoW that the order is a single order, and not part of a Merkle Tree.

Proof

Some services pick up new conditional orders automatically from on-chain events.

The proof data can be emitted on-chain, but it can also be retrieved from other supported on-chain services.

The location field signals where this data can be retrieved.

struct

Proof

{ uint256 location ; bytes data ; } note TheProof.location is intentionally not made anenum to allow for future extensibility as other proof locations may be integrated. Field Description location An integer representing the location where to find the proofs data location implementation specific data for retrieving the proofs

Locations

Name location data PRIVATE 0 bytes("") LOG 1 abi.encode(bytes[] order) whereorder = abi.encode(bytes32[] proof, ConditionalOrderParams params) SWARM 2 abi.encode(bytes32 swarmCac) WAKU 3 abi.encode(string protobufUri, string[] enrTreeOrMultiaddr, string contentTopic, bytes payload) IPFS 5 abi.encode(bytes32 ipfsCid) caution Locations above are for the point of defining a standard. The provided watch-tower currently doesnot support Merkle Tree proofs for orders. JSON schema for proofs It is expected that the proofs retrieved, excludingPRIVATE andLOG conform to a JSON schema:

{ "type" :

"object" , "properties" :

{ "proof" :

{ "type" :

"array" , "items" :

{ "type" :

"string" } } , "params" :

{ "type" :

"object" , "properties" :

{ "handler" :

{ "type" :

"string" } , "salt" :

{ "type" :

"string" } , "staticData" :

{ "type" :

"string" } } , "required" :

["handler" , "salt" , "staticData"] } , "offchainInput" :

{ "type" :

"string" } "description" :

{ "type" :

"string" } } , "required" :

["proof" , "params" ,] }

roots

Using anowner as a key, theroots mapping stores the Merkle Tree root for the conditional orders of thatowner .

mapping (address

=>

bytes32)

public roots ;

singleOrders

Usingowner, ctx as a key, thesingleOrders mapping stores the single orders for the conditional orders of thatowner .

mapping (address

=>

mapping (bytes32

=>

bool))

public singleOrders ;

cabinet

Usingowner, ctx as a key, thecabinet mapping stores the conditional order-specific data for the conditional orders of thatowner .

mapping (address

=>

mapping (bytes32

=>

bytes32))

public cabinet ;

swapGuards

Usingowner as a key, theswapGuards mapping stores the swap guards for the conditional orders of thatowner .

mapping (address

=> ISwapGuard)

public swapGuards ;

Functions

For users

setRoot

/setRootWithContext

Asafe orowner calls the respective setter method to set the Merkle Tree root for their conditional orders:

function

setRoot (bytes32 root , Proof calldata proof)

public ; function

setRootWithContext (bytes32 root , Proof calldata proof , IValueFactory factory , bytes

calldata data)

external ; Parameter Description root Merkle Tree root of conditional orders proofProof factory AnIValueFactory that will be used to populate thectx storage slot (if applicable) data Data to be passed to thefactory to populate thectx storage slot (if applicable) When a new merkle root is set, emitsMerkleRootSet(address indexed owner, bytes32 root, Proof proof) .

note ComposableCoW willNOT verify the proof data passed in via theproof parameter forsetRoot . It is the responsibility of the client and watch-tower to verify / validate this.

create

/createWithContext

Theowner calls the respective setter method to create a conditional order:

function

create (IConditionalOrder . ConditionalOrderParams calldata params , bool dispatch)

public ; function

createWithContext (IConditionalOrder . ConditionalOrderParams calldata params , IValueFactory factory , bytes

calldata data , bool dispatch)

external ; Parameter Description params ConditionalOrderParams factory AnIValueFactory that will be used to populate thectx storage slot (if applicable) data Data to be passed to thefactory to populate thectx storage slot (if applicable) dispatch Iftrue , broadcast theConditionalOrderCreated event

remove

Theowner calls theremove(bytes32 singleOrderHash) method to remove a conditional order:

function

remove (bytes32 singleOrderHash)

external ; Parameter Description singleOrderHash H(ConditionalOrderParams)

setSwapGuard

Theowner calls thesetSwapGuard(ISwapGuard guard) method to set a swap guard for a conditional order:

function

setSwapGuard (ISwapGuard swapGuard)

external ; Parameter Description swapGuard The swap guard contract

For watch-towers

getTradeableOrderWithSignature

A watch-tower calls thegetTradeableOrderWithSignature method to get a discrete order that is tradeable on CoW Protocol:

function

getTradeableOrderWithSignature (address owner , IConditionalOrder . ConditionalOrderParams calldata params , bytes

calldata offchainInput , bytes32 [ ]

calldata proof)

external

view

returns

(GPv2Order . Data memory order ,

bytes

memory signature) ; This function will:

1. Determine ifowner
2. is asafe
3. , and provide theSignatureVerifierMuxer
4. appropriate formatting for theERC-1271
5. signature submission to CoW Protocol.
6. If not asafe
7. , format theERC-1271
8. signature according toabi.encode(domainSeparator, staticData, offchainData)
9. .

Subsequently,ComposableCoW will:

1. Check that the order is authorized.
2. Check that the order type supports discrete order generation (i.e.IConditionalOrderGenerator
3.) by usingIERC165
4. (andrevert
5. if not, allowing the watch-tower to prune invalid monitored conditional orders).
6. CallgetTradeableOrder
7. on the handler to get the discrete order GPv2Order.Data
8.).
9. Generate the signing data as above.

Indexing

- ConditionalOrderCreated(address indexed owner, ConditionalOrderParams params)
- MerkleRootSet(address index owner, bytes32 root, Proof proof)

Custom error codes

- ProofNotAuthed()
-
    - the proof is not authorized (merkle root incorrect)
- SingleOrderNotAuthed()
-
    - the single order is not authorized
- SwapGuardRestricted()
-
    - the swap guard did not pass verification
- InvalidHandler()
-
    - the handler is not a valid conditional order
- InvalidFallbackHandler()
-
    - the fallback handler is not a valid conditional order

- InterfaceNotSupported()
-
    - the handler does not support theIConditionalOrder
- interface

Keep your orders watched A conditional order developerSHOULD use these error codes to ensure that the conditional order is well-formed and not garbage collected / rate limited by a watch-tower. * OrderNotValid(string) * - thestaticInput * parameters are not valid for the conditional order * PollTryNextBlock(string) * - signal to a watch-tower that polling should be attempted again * PollTryAtBlock(uint256 blockNumber, string) * - signal to a watch-tower that polling should be attempted again at a specific block number * PollTryAtEpoch(uint256 timestamp, string) * - signal to a watch-tower that polling should be attempted again at a specific epoch (unix timestamp) * PollNever(string) * - signal to a watch-tower that the conditional order should not be polled again (delete)

Off-chain

Watch-tower

As these orders are not automatically indexed by the CoW Protocol, there needs to be some method of relaying them to the Order Book API for inclusion in a batch.

This is the responsibility of awatch-tower . CoW Protocol runs a watch-tower that will monitor theConditionalOrderCreated event, and relay the discrete orders to the Order Book API.

There is also aDAppNode package for running a watch-tower. Edit this page Previous HooksTrampoline Next CoW AMM