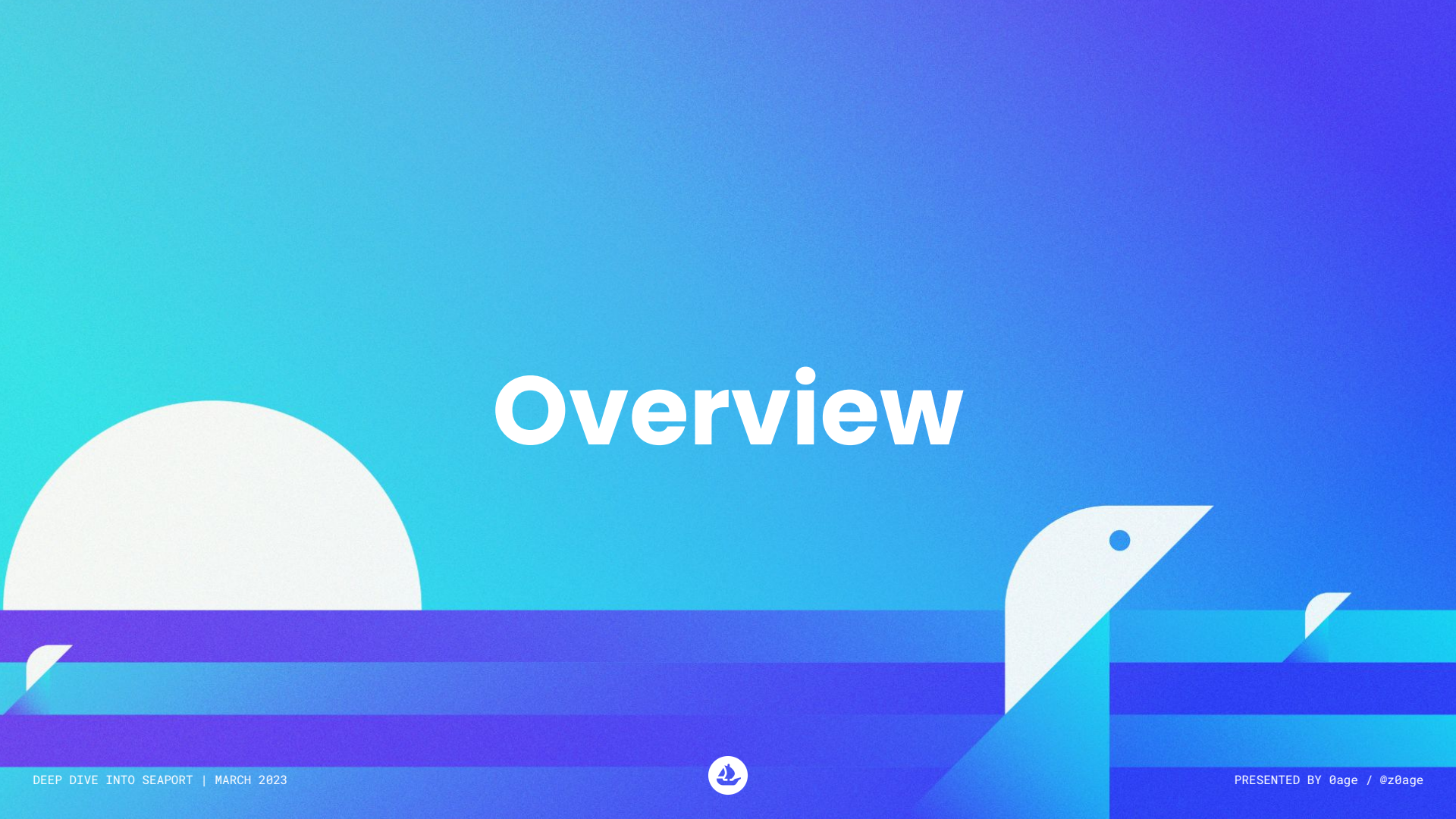
PRESENTED BY 0age / @z0age

Overview



Overview

- Seaport: marketplace protocol for safely & efficiently buying & selling NFTs
- Offer: the items (Native/ERC20/ERC721/ERC1155 tokens) you're willing to spend
- Consideration: the items you (or others) need to receive back
- Order: signed or validated listing, including the offer and the consideration
- Zone: external contract that validates restricted orders
- Conduit: external contract that performs token transfers
- Criteria-based item: ERC721/ERC1155 item that uses a merkle root in place of the token identifier (the caller specifies the identifier & provides a proof)
- Partial fill: order type that allows caller to specify a fraction to fill (where each item amount will be scaled down by that precise fraction)

Bulk Listings

PRESENTED BY 0age / @z0age

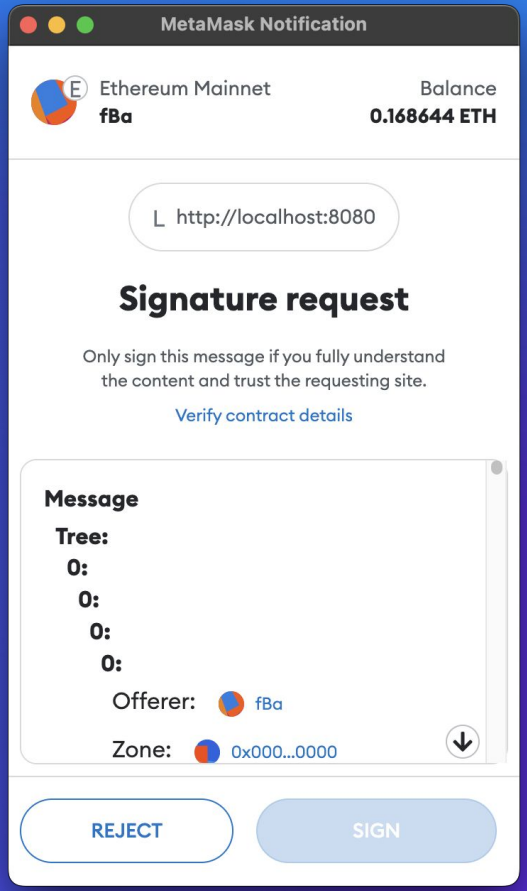
Bulk Listings

```
const orderType = {
  OrderComponents: [
    { name: "offerer", type: "address" },
    { name: "zone", type: "address" },
    { name: "offer", type: "OfferItem[]" },
    { name: "consideration", type: "ConsiderationItem[]" },
    { name: "orderType", type: "uint8" },
    { name: "startTime", type: "uint256" },
    { name: "endTime", type: "uint256" },
    { name: "zoneHash", type: "bytes32" },
    { name: "salt", type: "uint256" },
    { name: "conduitKey", type: "bytes32" },
    { name: "counter", type: "uint256" },
  ],
  OfferItem: [
    { name: "itemType", type: "uint8" },
    { name: "token", type: "address" },
    { name: "identifierOrCriteria", type: "uint256" },
    { name: "startAmount", type: "uint256" },
    { name: "endAmount", type: "uint256" },
  ],
  ConsiderationItem: [
    { name: "itemType", type: "uint8" },
    { name: "token", type: "address" },
    { name: "identifierOrCriteria", type: "uint256" },
    { name: "startAmount", type: "uint256" },
    { name: "endAmount", type: "uint256" },
    { name: "recipient", type: "address" },
  ],
};
```


Bulk Listings

```
const bulkOrderType = {
  BulkOrder: [{ name: "tree", type: "OrderComponents[2][2][2][2][2][2][2]" }],
  OrderComponents: [
    { name: "offerer", type: "address" },
    { name: "zone", type: "address" },
    { name: "offer", type: "OfferItem[]" },
    { name: "consideration", type: "ConsiderationItem[]" },
    { name: "orderType", type: "uint8" },
    { name: "startTime", type: "uint256" },
    { name: "endTime", type: "uint256" },
    { name: "zoneHash", type: "bytes32" },
    { name: "salt", type: "uint256" },
    { name: "conduitKey", type: "bytes32" },
    { name: "counter", type: "uint256" },
  ],
  OfferItem: [
    { name: "itemType", type: "uint8" },
    { name: "token", type: "address" },
    { name: "identifierOrCriteria", type: "uint256" },
    { name: "startAmount", type: "uint256" },
    { name: "endAmount", type: "uint256" },
  ],
  ConsiderationItem: [
    { name: "itemType", type: "uint8" },
    { name: "token", type: "address" },
    { name: "identifierOrCriteria", type: "uint256" },
    { name: "startAmount", type: "uint256" },
    { name: "endAmount", type: "uint256" },
    { name: "recipient", type: "address" },
  ],
};
```


Bulk Listings



Bulk Listings

```
// ECDSA Signature (64 or 65 bytes) + key (3 bytes) + proof (32 * height bytes)
0x
10101010101010101010101010101010101010101010101010101010101010
10101010101010101010101010101010101010101010101010101010101010
000005
20202020202020202020202020202020202020202020202020202020202020
30303030303030303030303030303030303030303030303030303030303030
40404040404040404040404040404040404040404040404040404040404040
50505050505050505050505050505050505050505050505050505050505050
60606060606060606060606060606060606060606060606060606060606060
70707070707070707070707070707070707070707070707070707070707070
80808080808080808080808080808080808080808080808080808080808080
```


Bulk Listings

```
function _verifySignature(
    address offerer,
    bytes32 orderHash,
    bytes memory signature
) internal view {
    // ... snip ... //

    // Derive the EIP-712 domain separator.
    bytes32 domainSeparator = _domainSeparator();

    // Derive original EIP-712 digest using domain separator and order hash.
    bytes32 originalDigest = _deriveEIP712Digest(
        domainSeparator,
        orderHash
    );

    // Read the length of the signature from memory and place on the stack.
    uint256 originalSignatureLength = signature.length;

    // Determine effective digest if signature has a valid bulk order size.
    bytes32 digest;
    if (_isValidBulkOrderSize(originalSignatureLength)) {
        // Rederive order hash and digest using bulk order proof.
        (orderHash) = _computeBulkOrderProof(signature, orderHash);
        digest = _deriveEIP712Digest(domainSeparator, orderHash);
    } else {
        // Supply the original digest as the effective digest.
        digest = originalDigest;
    }

    // Ensure that the signature for the digest is valid for the offerer.
    _assertValidSignature(
        offerer,
        digest,
        originalDigest,
        originalSignatureLength,
        signature
    );
}
```


Bulk Listings

```
/**
 * @dev Determines whether the specified bulk order size is valid.
 *
 * @param signature The signature of the bulk order to check.
 *
 * @return validLength True if bulk order size is valid, false otherwise.
 */
function _isValidBulkOrderSize(
    bytes memory signature
) internal pure returns (bool validLength) {
    validLength =
        signature.length < 837 &&
        signature.length > 98 &&
        ((signature.length - 67) % 32) < 2;
}
```


Bulk Listings

```solidity
/**
 * @dev Computes the bulk order hash for the specified proof and leaf. Note
 *      that if an index that exceeds the number of orders in the bulk order
 *      payload will instead "wrap around" and refer to an earlier index.
 *
 * @param proofAndSignature The proof and signature of the bulk order.
 * @param leaf              The leaf of the bulk order tree.
 *
 * @return bulkOrderHash The bulk order hash.
 * @return signature     The signature of the bulk order.
 */
function _computeBulkOrderProof(
    bytes memory proofAndSignature,
    bytes32 leaf
) internal view returns (bytes32 bulkOrderHash, bytes memory signature) {
    bytes32 root = leaf;

    // proofAndSignature with odd length is a compact signature (64 bytes).
    uint256 length = proofAndSignature.length % 2 == 0 ? 65 : 64;

    // Create a new array of bytes equal to the length of the signature.
    signature = new bytes(length);

    // Iterate over each byte in the signature.
    for (uint256 i = 0; i < length; ++i) {
        // Assign the byte from the proofAndSignature to the signature.
        signature[i] = proofAndSignature[i];
    }

    // Compute the key by extracting the next three bytes from the
    // proofAndSignature.
    uint256 key = (((uint256(uint8(proofAndSignature[length])) << 16) |
        ((uint256(uint8(proofAndSignature[length + 1]))) << 8)) |
        (uint256(uint8(proofAndSignature[length + 2]))));

    uint256 height = (proofAndSignature.length - length) / 32;

    // Create an array of bytes32 to hold the proof elements.
    bytes32[] memory proofElements = new bytes32[](height);
```

```solidity
    // Iterate over each proof element.
    for (uint256 elementIndex = 0; elementIndex < height; ++elementIndex) {
        // Compute the starting index for the current proof element.
        uint256 start = (length + 3) + (elementIndex * 32);

        // Create a new array of bytes to hold the current proof element.
        bytes memory buffer = new bytes(32);

        // Iterate over each byte in the proof element.
        for (uint256 i = 0; i < 32; ++i) {
            // Assign the byte from the proofAndSignature to the buffer.
            buffer[i] = proofAndSignature[start + i];
        }

        // Decode the current proof element from the buffer and assign it to
        // the proofElements array.
        proofElements[elementIndex] = abi.decode(buffer, (bytes32));
    }

    // Iterate over each proof element.
    for (uint256 i = 0; i < proofElements.length; ++i) {
        // Retrieve the proof element.
        bytes32 proofElement = proofElements[i];

        // Check if the current bit of the key is set.
        if ((key >> i) % 2 == 0) {
            // If the current bit is not set, then concatenate the root and
            // the proof element, and compute the keccak256 hash of the
            // concatenation to assign it to the root.
            root = keccak256(abi.encodePacked(root, proofElement));
        } else {
            // If the current bit is set, then concatenate the proof element
            // and the root, and compute the keccak256 hash of the
            // concatenation to assign it to the root.
            root = keccak256(abi.encodePacked(proofElement, root));
        }
    }

    // Compute the bulk order hash and return it.
    bulkOrderHash = keccak256(
        abi.encodePacked(_bulkOrderTypehashes[height], root)
    );

    // Return the signature.
    return (bulkOrderHash, signature);
```


Zones

Zones

```
/**
 * @title  ZoneInterface
 * @notice Contains functions exposed by a zone.
 */
interface ZoneInterface {
    /**
     * @dev Validates an order.
     *
     * @param zoneParameters The context about the order fulfillment and any
     *                       supplied extraData.
     *
     * @return validOrderMagicValue The magic value that indicates a valid
     *                              order.
     */
    function validateOrder(
        ZoneParameters calldata zoneParameters
    ) external returns (bytes4 validOrderMagicValue);

    /**
     * @dev Returns the metadata for this zone.
     *
     * @return name The name of the zone.
     * @return schemas The schemas that the zone implements.
     */
    function getSeaportMetadata()
        external
        view
        returns (
            string memory name,
            Schema[] memory schemas // map to Seaport Improvement Proposal IDs
        );
}
```

```
/**
 * @dev Restricted orders are validated post-execution by calling validateOrder
 *      on the zone. This struct provides context about the order fulfillment
 *      and any supplied extraData, as well as all order hashes fulfilled in a
 *      call to a match or fulfillAvailable method.
 */
struct ZoneParameters {
    bytes32 orderHash;
    address fulfiller;
    address offerer;
    SpentItem[] offer;
    ReceivedItem[] consideration;
    bytes extraData;
    bytes32[] orderHashes;
    uint256 startTime;
    uint256 endTime;
    bytes32 zoneHash;
}

/**
 * @dev A spent item is translated from a utilized offer item and has four
 *      components: an item type (ETH or other native tokens, ERC20, ERC721, and
 *      ERC1155), a token address, a tokenId, and an amount.
 */
struct SpentItem {
    ItemType itemType;
    address token;
    uint256 identifier;
    uint256 amount; // <== UNRELIABLE DURING ZONE CHECK ON v1.4!!!
}

/**
 * @dev A received item is translated from a utilized consideration item and has
 *      the same four components as a spent item, as well as an additional fifth
 *      component designating the required recipient of the item.
 */
struct ReceivedItem {
    ItemType itemType;
    address token;
    uint256 identifier;
    uint256 amount;
    address payable recipient;
}
```

Zones

```
/**
 * @dev An event that is emitted when a SIP-5 compatible contract is deployed.
 */
event SeaportCompatibleContractDeployed();

/**
 * @dev Zones and contract offerers can communicate which schemas they implement
 *      along with any associated metadata related to each schema.
 */
struct Schema {
    uint256 id; /// Seaport Improvement Proposal (SIP) ID
    bytes metadata; /// Optional additional metadata
}

/**
 * @dev Returns Seaport metadata for this contract, returning the
 *      contract name and supported schemas.
 *
 * @return name    The contract name
 * @return schemas The supported SIPs
 */
function getSeaportMetadata() external view returns (
    string memory name,
    Schema[] memory schemas
);
```

ProjectOpenSea / SIPs

Zones

If `extraData` is supplied as part of a SIP-6-compliant order, it MUST be prefixed with a `version` byte that follows the format in the below table.

version byte	description	decoding scheme	fixed data hashing scheme
0x00	single variable data array	`abi.decode(extraData[1:], (bytes))`	n/a
0x01	single fixed data array	`abi.decode(extraData[1:], (bytes))`	`keccak256(fixedDataArray)`
0x02	single variable data array and single fixed data array	`abi.decode(extraData[1:], (bytes, bytes))`	`keccak256(fixedDataArray)`
0x03	multiple variable data arrays	`abi.decode(extraData[1:], (bytes[]))`	n/a
0x04	multiple fixed data arrays	`abi.decode(extraData[1:], (bytes[]))`	`keccak256(abi.encode(keccak256(fixedDataArrays[0]), keccak256(fixedDataArrays[1]), ...))`
0x05	multiple variable data arrays and multiple fixed data arrays	`abi.decode(extraData[1:], (bytes[], bytes[]))`	`keccak256(abi.encode(keccak256(fixedDataArrays[0]), keccak256(fixedDataArrays[1]), ...))`

Contract-based Listings

PRESENTED BY 0age / @z0age

Contract-based Listings

```
interface ContractOffererInterface {
    /**
     * @dev Generates an order with the specified minimum and maximum spent
     *      items, and optional context (supplied as extraData).
     *
     * @param fulfiller       The address of the fulfiller.
     * @param minimumReceived The minimum items that the caller is willing to
     *                        receive.
     * @param maximumSpent    The maximum items the caller is willing to spend.
     * @param context         Additional context of the order.
     *
     * @return offer         A tuple containing the offer items.
     * @return consideration A tuple containing the consideration items.
     */
    function generateOrder(
        address fulfiller,
        SpentItem[] calldata minimumReceived,
        SpentItem[] calldata maximumSpent,
        bytes calldata context // encoded based on the schemaID
    )
        external
        returns (SpentItem[] memory offer, ReceivedItem[] memory consideration);

    /**
     * @dev Ratifies an order with the specified offer, consideration, and
     *      optional context (supplied as extraData).
     *
     * @param offer         The offer items.
     * @param consideration The consideration items.
     * @param context       Additional context of the order.
     * @param orderHashes   The hashes to ratify.
     * @param contractNonce The nonce of the contract.
     *
     * @return ratifyOrderMagicValue The magic value returned by the contract
     *                               offerer.
     */
    function ratifyOrder(
        SpentItem[] calldata offer,
        ReceivedItem[] calldata consideration,
        bytes calldata context, // encoded based on the schemaID
        bytes32[] calldata orderHashes,
        uint256 contractNonce
    ) external returns (bytes4 ratifyOrderMagicValue);
```

```
/**
 * @dev View function to preview an order generated in response to a minimum
 *      set of received items, maximum set of spent items, and context
 *      (supplied as extraData).
 *
 * @param caller          The address of the caller (e.g. Seaport).
 * @param fulfiller       The address of the fulfiller (e.g. the account
 *                        calling Seaport).
 * @param minimumReceived The minimum items that the caller is willing to
 *                        receive.
 * @param maximumSpent    The maximum items the caller is willing to spend.
 * @param context         Additional context of the order.
 *
 * @return offer         A tuple containing the offer items.
 * @return consideration A tuple containing the consideration items.
 */
function previewOrder(
    address caller,
    address fulfiller,
    SpentItem[] calldata minimumReceived,
    SpentItem[] calldata maximumSpent,
    bytes calldata context // encoded based on the schemaID
)
    external
    view
    returns (SpentItem[] memory offer, ReceivedItem[] memory consideration);

/**
 * @dev Gets the metadata for this contract offerer.
 *
 * @return name    The name of the contract offerer.
 * @return schemas The schemas supported by the contract offerer.
 */
function getSeaportMetadata()
    external
    view
    returns (
        string memory name,
        Schema[] memory schemas // map to Seaport Improvement Proposal IDs
    );

// Additional functions and/or events based on implemented schemaIDs
```


Contract-based Listings

```
{
    address offerer = orderParameters.offerer;
    bool success;
    (MemoryPointer cdPtr, uint256 size) = _encodeGenerateOrder(
        orderParameters,
        context
    );
    assembly {
        success := call(gas(), offerer, 0, cdPtr, size, 0, 0)
    }

    {
        // Note: overflow impossible; nonce can't increment that high.
        uint256 contractNonce;
        unchecked {
            // Note: nonce will be incremented even for skipped orders,
            // and  even if generateOrder's return data does not satisfy
            // all the constraints. This is the case when errorBuffer
            // != 0 and revertOnInvalid == false.
            contractNonce = _contractNonces[offerer]++;
        }

        assembly {
            // Shift offerer address up 96 bytes and combine with nonce.
            orderHash := xor(
                contractNonce,
                shl(ContractOrder_orderHash_offerer_shift, offerer)
            )
        }
    }

    // Revert or skip if the call to generate the contract order failed.
    if (!success) {
        return _revertOrReturnEmpty(revertOnInvalid, orderHash);
    }
}
```

```
{
    // Designate lengths.
    uint256 originalOfferLength = orderParameters.offer.length;
    uint256 newOfferLength = offer.length;

    // Explicitly specified offer items cannot be removed.
    if (originalOfferLength > newOfferLength) {
        _revertInvalidContractOrder(orderHash);
    }

    // Iterate over each specified offer (e.g. minimumReceived) item.
    for (uint256 i = 0; i < originalOfferLength; ) {…
    }

    // Assign the returned offer item in place of the original item.
    orderParameters.offer = offer;
}

{
    // Designate lengths & memory locations.
    ConsiderationItem[] memory originalConsiderationArray = (
        orderParameters.consideration
    );
    uint256 newConsiderationLength = consideration.length;

    // New consideration items cannot be created.
    if (newConsiderationLength > originalConsiderationArray.length) {
        _revertInvalidContractOrder(orderHash);
    }

    // Iterate over returned consideration & do not exceed maximumSpent.
    for (uint256 i = 0; i < newConsiderationLength; ) {…
    }

    // Assign returned consideration item in place of the original item.
    orderParameters.consideration = consideration;
}

// Revert if any item comparison failed.
if (errorBuffer != 0) {
    _revertInvalidContractOrder(orderHash);
}

// Return order hash and full fill amount (numerator & denominator = 1).
return (orderHash, 1, 1);
```


Optimizations

Optimizations

```
/**
 * @dev Takes a bytes array in memory and copies it to a new location in
 *      memory.
 *
 * @param src A memory pointer referencing the bytes array to be copied (and
 *            pointing to the length of the bytes array).
 * @param src A memory pointer referencing the location in memory to copy
 *            the bytes array to (and pointing to the length of the copied
 *            bytes array).
 *
 * @return size The size of the bytes array.
 */
function _encodeBytes(
    MemoryPointer src,
    MemoryPointer dst
) internal view returns (uint256 size) {
    unchecked {
        // Mask the length of the bytes array to protect against overflow
        // and round up to the nearest word.
        // Note: `size` also includes the 1 word that stores the length.
        size = (src.readUint256() + SixtyThreeBytes) & OnlyFullWordMask;

        // Copy the bytes array to the new memory location.
        src.copy(dst, size);
    }
}
```

d1ll0n / abi-lity

```
/**
 * @dev Takes an offer array from calldata and copies it into memory.
 *
 * @param cdPtrLength A calldata pointer to the start of the offer array
 *                    in calldata which contains the length of the array.
 *
 * @return mPtrLength A memory pointer to the start of the offer array in
 *                    memory which contains the length of the array.
 */
function _decodeOffer(
    CalldataPointer cdPtrLength
) internal pure returns (MemoryPointer mPtrLength) {
    assembly {
        // Retrieve length of array, masking to prevent potential overflow.
        let arrLength := and(calldataload(cdPtrLength), OffsetOrLengthMask)

        // Get the current free memory pointer.
        mPtrLength := mload(FreeMemoryPointerSlot)

        // Write the array length to memory.
        mstore(mPtrLength, arrLength)

        // Derive the head by adding one word to the length pointer.
        let mPtrHead := add(mPtrLength, OneWord)

        // Derive the tail by adding one word per element (note that structs
        // are written to memory with an offset per struct element).
        let mPtrTail := add(mPtrHead, shl(OneWordShift, arrLength))

        // Track the next tail, beginning with the initial tail value.
        let mPtrTailNext := mPtrTail

        // Copy all offer array data into memory at the tail pointer.
        calldatacopy(
            mPtrTail,
            add(cdPtrLength, OneWord),
            mul(arrLength, OfferItem_size)
        )

        // Track the next head pointer, starting with initial head value.
        let mPtrHeadNext := mPtrHead

        // Iterate over each head pointer until it reaches the tail.
        for {} lt(mPtrHeadNext, mPtrTail) {} {
            // Write the next tail pointer to next head pointer in memory.
            mstore(mPtrHeadNext, mPtrTailNext)

            // Increment the next head pointer by one word.
            mPtrHeadNext := add(mPtrHeadNext, OneWord)

            // Increment the next tail pointer by the size of an offer item.
            mPtrTailNext := add(mPtrTailNext, OfferItem_size)
        }

        // Update free memory pointer to allocate memory up to end of tail.
        mstore(FreeMemoryPointerSlot, mPtrTailNext)
    }
}
```


Optimizations

```
/**
 * @dev Converts a function taking a calldata pointer and returning a memory
 *      pointer into a function taking that calldata pointer and returning
 *      an AdvancedOrder type.
 *
 * @param inFn The input function, taking an arbitrary calldata pointer and
 *             returning an arbitrary memory pointer.
 *
 * @return outFn The output function, taking an arbitrary calldata pointer
 *               and returning an AdvancedOrder type.
 */
function _toAdvancedOrderReturnType(
    function(CalldataPointer) internal pure returns (MemoryPointer) inFn
)
    internal
    pure
    returns (
        function(CalldataPointer)
            internal
            pure
            returns (AdvancedOrder memory) outFn
    )
{
    assembly {
        outFn := inFn
    }
}
```

```
/**
 * @notice Fulfill an order with an arbitrary number of items for offer and
 *         consideration. Note that this function does not support
 *         criteria-based orders or partial filling of orders (though
 *         filling the remainder of a partially-filled order is supported).
 *
 * @custom:param order                The order to fulfill. Note that both the
 *                                    offerer and the fulfiller must first approve
 *                                    this contract (or the corresponding conduit if
 *                                    indicated) to transfer any relevant tokens on
 *                                    their behalf and that contracts must implement
 *                                    `onERC1155Received` to receive ERC1155 tokens
 *                                    as consideration.
 * @param fulfillerConduitKey A bytes32 value indicating what conduit, if
 *                                    any, to source the fulfiller's token approvals
 *                                    from. The zero hash signifies that no conduit
 *                                    should be used (and direct approvals set on
 *                                    this contract).
 *
 * @return fulfilled A boolean indicating whether the order has been
 *                   successfully fulfilled.
 */
function fulfillOrder(
    /**
     * @custom:name order
     */
    Order calldata,
    bytes32 fulfillerConduitKey
) external payable override returns (bool fulfilled) {
    // Convert order to "advanced" order, then validate and fulfill it.
    fulfilled = _validateAndFulfillAdvancedOrder(
        _toAdvancedOrderReturnType(_decodeOrderAsAdvancedOrder)(
            CalldataStart.pptr()
        ),
        new CriteriaResolver[](0), // No criteria resolvers supplied.
        fulfillerConduitKey,
        msg.sender
    );
}
```


https://discord.gg/mFANYTUn