Alembic NFT minting contracts

This repository is a collection of various NFT minting contracts templates you can freely use for your own needs.

Documentation: https://alembic-tech.github.io/nft-minting-contracts/
PDF version: https://alembic-tech.github.io/nft-minting-contracts/ebook.pdf

Table of Contents

ERC721 Minting Contracts 1
Whitelist with Merkle Tree 1
ERC1155 Minting Contracts 3
Controlled by EIP-712 signatures 3

ERC721 Minting Contracts

Whitelist with Merkle Tree

ERC721MerkleTreeMinter is a minter contract expected to work in tandem with almost any ERC721 contract which implements the IERC721Mintable interface.

The `IERC721Mintable` interface is pretty basic and a typical ERC721 mint function:

```
function mint(address to) external;
```

A sample ERC721 contract, ERC721Test is provided, based on OpenZeppelin roles, and restricting mints to only contracts having the `MINTER_ROLE`.

ERC721MerkleTreeMinter.spec.ts contains examples on how to use `ERC721MerkleTreeMinter`.

The `ERC721MerkleTreeMinter` contract is initialized with a Merkle Tree root, computed from leaves having the following structure:

```
const leafSignature: LeafSignature = [
    { type: "address", name: "address" }, ①
    { type: "uint256", name: "quantity" }, ②
    { type: "uint256", name: "nonce" }, ③
]
```

① Address of the whitelisted user

② Maximum quantity of NFTs the whitelisted user is allowed to mint

③ A nonce allowing to have a user included more than once in the tree if need be

Initialize your Merkle Tree with an array of `LeafSourceObject`:

```
// compute merkle tree with user1 in the whitelist twice with different quantities
const leaves: LeafSourceObject[] = [
    { address: user1.address, quantity: 1, nonce: 123 },
    { address: user1.address, quantity: 2, nonce: 456 },
]
```

Then compute the Merkle Tree:

You will need to keep track of the merkle proofs for each user address in the exact same order of your leaves array.
It is quite common to either embed a JSON file in your dApp or store such data a database and have your dApp fetch the relevant data for the connected user.

```
// compute merkle tree
const merkleTree = new WMerkleTree(leaves, leafSignature)
const merkleRoot = merkleTree.getHexRoot()
// return user merkle tree proofs for each leaf
return leaves.map((leaf, index) => merkleTree.getHexProof(index))
```

The `deployWhitelistedMinter` and `deployMinterWithLeaves` functions included in the test suite are handy if you want to do this in your scripts.

Finally here is an example of call to have your user mint his NFTs:

```
const tx = await minter.connect(user1).mint(
    user1.address, ①
    1, ②
    1, ③
    123, ④
    proofs[0], ⑤
    { value: price } ⑥
)
```

① address of the user

② quantity to mint

③ max quantity mintable by that user

④ the nonce

⑤ the merkle tree proofs

⑥ purchase price

ERC1155 Minting Contracts

Controlled by EIP-712 signatures

ERC1155VoucherMinter is a minter contract expected to work in tandem with almost any ERC1155 contract which implements the IERC1155Mintable interface.

This minting approach is particularly useful when you have some kind of backends controlling a wallet allowed to sign `ERC1155VoucherMinter.MintRequest`.

The `IERC1155Mintable` interface is pretty basic and a typical ERC1155 mint function:

```
function mint(
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) external;
```

A sample ERC1155 contract, ERC1155Test is provided, based on OpenZeppelin roles, and restricting mints to only contracts having the `MINTER_ROLE`.

ERC1155VoucherMinter.spec.ts contains examples on how to use `ERC1155VoucherMinter`.

From your backend, sign a mint request, using the provided getTypedRequest function.

```
// sign request to mint
const { domain, types, request } = await getTypedRequest(minter.address, user.address,
tokenId, quantity, nonce)
const signature = await operator._signTypedData(domain, types, request)
```

In your dApp, to have the user to send the mint transaction, using Ethers:

Such transactions can be relayed if you want.
Any wallet can send this transaction.

```
const { domain, types, request } = await getTypedRequest(minter.address, user.address,
tokenId, quantity, nonce)
// TODO: make your backend provide the previously generated `signature`
const tx = await minter.mint({ ...request, signature })
```