Web3 Unleashed: Write an NFT Smart Contract with Royalties¶

Written byEmily Lin

Last updated 8/5/2022

Overview¶

In this chapter of Web3 Unleashed, we're going to briefly go over what an Ethereum Improvement Proposal (EIP) and an Ethereum Request for Comment (ERC) is, how they are used, and implement the ERC-2981, NFT royalty standard, as an example.

What is an EIP?¶

EIP stands for Ethereum Improvement Proposal, a technical design document that specifies new features and processes for Ethereum.

EIPs serve as the primary mechanism for

1. Proposing new features
2. Collecting community technical input on an issue
3. Documenting the design decisions that have gone into Ethereum.

There are three types of EIPs:

1. Standard Track
2. describes any change that affects most or all Ethereum implementations. This is further split into the categoriesCore
3. ,Networking
4. ,Interface
5. , andERC
6. .* Core
7.
    - describes improvements that either require a consensus fork, a major technical change to the "rules" of Ethereum (how gas is charged inEIP 5
8.
    -), or are generally relevant to core developer discussions (miners checking that gas price is sufficiently high inEIP-86
9.
    -).
10.
    - Networking
11.
    - describes improvements arounddevp2p
12.
    - ,Light Ethereum Subprotocol
13.
    - , and the network protocol specifications of whisper andswarm
14.
    - .
15.
    - Interface
16.
    - describes improvements on API/RPC specifications and standards and certain language-level standards like method names and contract ABIs.
17.
    - ERC
18.
    - describes application level standards. We'll dive more in depth in the next section.
19. Meta
20. describes changing a process surrounding Ethereum, such as changes to a decision-making process.
21. Informational
22. provides general information or describes an Ethereum design issue, but does not propose a new feature. Users are free to ignore informational EIPs because informational EIPs do not necessarily represent an Ethereum community recommendation.

What is an ERC?¶

ERC stands for Ethereum Request for Comment. As mentioned above, this describes application level conventions such as token standards, name registries, URI schemes, library/package formats, and wallet formats. ERCs specify a required set of functions that contracts need to implement so that apps and other contracts can understand how to interact with them. For example, one of the most popular standards is theERC-721 standard, which defines what an NFT is. Because an application knows what an ERC-721 looks like, it knows what functions and properties it can interact with on the contract.

Note that ERCs are not considered a core EIP, so adopting the standard is up to the developer. As a result, raising awareness around an ERC is crucial to its utility and success.

Why do EIPs and ERCs matter?¶

EIPs serve as the core way in which governance happens in Ethereum. Anyone is allowed to propose them, and community members comment, debate, and collaborate to decide whether or not it should be adopted! You can find the guidelines to submit your ownhere .

ERCs are what powers the composability of smart contracts! Composability defines the ability of dapps and contracts to interact with each other. For example, the ERC-2981 NFT royalty standard defines how royalty information is stored on the contract, so that when dapps such as marketplaces make sales, they know how to get the royalty information necessary to compensate the artist!

What's in an ERC-2981?¶

As mentioned above, an ERC-2981 is the royalty standard. In order to qualify as an ERC-2981, the smart contract must have the following functions:

pragma

solidity

^ 0.6.0 ; import

"./IERC165.sol" ; /// /// @dev Interface for the NFT Royalty Standard /// interface

IERC2981

is

IERC165

{

/// ERC165 bytes to add to interface array - set in parent contract

/// implementing this standard

///

/// bytes4(keccak256("royaltyInfo(uint256,uint256)")) == 0x2a55205a

/// bytes4 private constant _INTERFACE_ID_ERC2981 = 0x2a55205a;

/// _registerInterface(_INTERFACE_ID_ERC2981);

/// @notice Called with the sale price to determine how much royalty

// is owed and to whom.

/// @param _tokenId - the NFT asset queried for royalty information

/// @param _salePrice - the sale price of the NFT asset specified by _tokenId

/// @return receiver - address of who should be sent the royalty payment

/// @return royaltyAmount - the royalty payment amount for _salePrice

function

royaltyInfo (

uint256

_tokenId ,

uint256

_salePrice

)

external

view

returns

(

address

receiver ,

uint256

royaltyAmount

); } interface

IERC165

{

/// @notice Query if a contract implements an interface

/// @param interfaceID The interface identifier, as specified in ERC-165

/// @dev Interface identification is specified in ERC-165. This function

/// uses less than 30,000 gas.

/// @return true if the contract implements interfaceID and

/// interfaceID is not 0xffffffff, false otherwise

function

supportsInterface (bytes4

interfaceID)

external

view

returns

(bool); } Side note: ERC-165 is a standard that allows contracts to declare their support of an interface. This is what will allow marketplaces to check if the NFT supports the royalty standard! That might look something like this in a marketplace contract:

bytes4

private

constant

_INTERFACE_ID_ERC2981

=

0x2a55205a ; function

checkRoyalties (address

_contract)

internal

returns

(bool)

{

(bool

success)

=

IERC165 (_contract). supportsInterface (_INTERFACE_ID_ERC2981);

return

success ; } Before we start writing code, let's first step through some important caveats about the NFT royalty standard.

1. The royalty payments arenot
2. enforced by the standard. It is up to the marketplace to act upon that information. Currently, Coinbase NFT, Rarible, SuperRare, and Zora pay out royalties for an ERC-2981. If your NFT is being sold on OpenSea, you will have to separately set royalties on your NFT through their website.
3. This standard does not requireERC-721
4. (NFT standard) andERC-1155
5. (multi-token standard) compatibility. So it is perhaps more appropriate to say a "universal royalty standard"!

Let's Write an ERC-2981

Now that we've covered what EIPs are, what ERCs are, and how they represent the ERC-2981, let's actually write an ERC-721 NFT smart contract that implements the ERC-2981 royalty standard. You can find the completed codehere . We'll be importing Open Zeppelin's contracts, which provide secure, pre-written implementations of the ERC that our contract can just inherit!

Note that we will not be covering the basics of the ERC-721 standard. You can find a great Infura blog detailing what it is and how to implement ithere .

Download System Requirements¶

You'll need to install:

- Node.js
- , v12 or higher
- truffle
- ganache UI
- organache CLI

Create an Infura account and project¶

To connect your DApp to Ethereum mainnet and testnets, you'll need an Infura account. Sign up for an accounthere .

Once you're signed in, create a project! Let's call itnft-royalty , and select Web3 API from the dropdown

Register for a MetaMask wallet¶

To interact with your DApp in the browser, you'll need a MetaMask wallet. Sign up for an accounthere .

Download VS Code¶

Feel free to use whatever IDE you want, but we highly recommend using VS Code! You can run through most of this tutorial using the Truffle extension to create, build, and deploy your smart contracts, all without using the CLI! You can read more about ithere .

Get Some Test Eth¶

In order to deploy to the public testnets, you'll need some test Eth to cover your gas fees has a great MultiFaucet

that deposits funds across 8 different networks all at once.

Set Up Your Project¶

Truffle has some nifty functions to scaffold your truffle project and add example contracts and tests. We'll be building our project in a folder callednft-royalty .

truffle init nft-royaltycd

nft-royalty truffle create contract RoyalPets truffle create test

TestRoyalties Afterwards, your project structure should look something like this:

nft-royalty ├─── contracts | └─── RoyalPets.sol ├─── migrations | └─── 1_deploy_contracts.js ├─── test | └─── test_royalties.js └─── truffle-config.js

Write the NFT Smart Contract¶

Open Zeppelin already provides secure, pre-written ERC-2981 and ERC-721 contract implementations we can just inherit! To download them, simply callnpm i "@openzeppelin/contracts" .

With OpenZeppelin, we have a few ways of identifying that an NFT contract fits the royalty standard. Since our base contract will be an ERC-721, we have the option of inheriting OpenZeppelin's royalty extensionERC721Royalty . This contract overrides the_burn function to also clear the royalty information for the token.

Important note! Both this function and the_burn function from OpenZeppelin do not check fortokenId ownership. That means anyone can burn this NFT. If you want to avoid this, add arequire check checking for that condition.

function

_burn (uint256

tokenId)

internal

virtual

override

{

super . _burn (tokenId);

_resetTokenRoyalty (tokenId); } Upon contract creation, we want to set a defaulty royalty recipient and percentage. Note that OpenZeppelin calculates the royalty fee using basis points. In order to set the default recipient to be the owner of the contract and the fee to be 1%, set it in the constructor:

import

"@openzeppelin/contracts/token/ERC721/extensions/ERC721Royalty.sol" ; contract

RoyalPets

is

ERC721Royalty

{

constructor ()

ERC721 ("RoyalPets" ,

"RP")

{

_setDefaultRoyalty (msg . sender ,

100);

} } However, in this tutorial, we want to use OpenZeppelin'sERC721URIStorage extension. In this case, we want it to also

inherit theproperties of OpenZeppelin'sERC2981 contract like so:

import

"@openzeppelin/contracts/token/common/ERC2981.sol" ; import

"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol" ; contract

RoyalPets

is

ERC721URIStorage ,

ERC2981

{

constructor ()

ERC721 ("RoyalPets" ,

"RP")

{

_setDefaultRoyalty (msg . sender ,

100);

} } We do run into a problem though.ERC721URIStorage andERC2981 both overridesupportsInterface ! To fix this, we need to override it inRoyalPets as well. Add in this function:

function

supportsInterface (bytes4

interfaceId)

public

view

virtual

override (ERC721 ,

ERC2981)

returns

(bool)

{

return

super . supportsInterface (interfaceId); } Additionally, because we are no longer inheritingERC721Royalty , we no longer have its_burn override. Let's add that in:

function

_burn (uint256

tokenId)

internal

virtual

override

{

super . _burn (tokenId);

_resetTokenRoyalty (tokenId); } So that external accounts can burn their NFTs, expose a public version of burn:

function

burnNFT (uint256

tokenId)

public

{

_burn (tokenId); } Lastly, we'll add in the actual NFT minting functionality. We'll create two types of minting functions: one that mints tokens with the default royalty info, and one that specifies royalty info on a per-token basis.

As mentioned before, those basics are covered in this Infura blog . One minor difference is that we will not be using a static metadata file to populate thetokenURI . The two minting functions look like this:

function

mintNFT (address

recipient ,

string

memory

tokenURI)

public

onlyOwner

returns

(uint256)

{

_tokenIds . increment ();

uint256

newItemId

=

_tokenIds . current ();

_safeMint (recipient ,

newItemId);

_setTokenURI (newItemId ,

tokenURI);

return

newItemId ; } function

mintNFTWithRoyalty (address

recipient ,

string

memory

tokenURI ,

address

royaltyReceiver ,

uint96

feeNumerator)

public

onlyOwner

returns

(uint256)

{

uint256

tokenId

=

mintNFT (recipient ,

tokenURI);

_setTokenRoyalty (tokenId ,

royaltyReceiver ,

feeNumerator);

return

tokenId ; } Your final smart contract should look like this:

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; import

"@openzeppelin/contracts/token/common/ERC2981.sol" ; import

"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol" ; import

"@openzeppelin/contracts/access/Ownable.sol" ; import

"@openzeppelin/contracts/utils/Counters.sol" ; contract

RoyalPets

is

ERC721URIStorage ,

ERC2981 ,

Ownable

{

using

Counters

for

Counters . Counter ;

```
Counters . Counter
private
_tokenIds ;
constructor ()
ERC721 ( "RoyalPets" ,
"RP" )
{
_setDefaultRoyalty ( msg . sender ,
100 );
}
function
supportsInterface ( bytes4
interfaceId )
public
view
virtual
override ( ERC721 ,
ERC2981 )
returns
( bool )
{
return
super . supportsInterface ( interfaceId );
}
function
_burn ( uint256
tokenId )
internal
virtual
override
{
super . _burn ( tokenId );
_resetTokenRoyalty ( tokenId );
}
function
burnNFT ( uint256
tokenId )
```

```
public
onlyOwner
{
_burn ( tokenId );
}
function
mintNFT ( address
recipient ,
string
memory
tokenURI )
public
onlyOwner
returns
( uint256 )
{
_tokenIds . increment ();
uint256
newItemId
=
_tokenIds . current ();
_safeMint ( recipient ,
newItemId );
_setTokenURI ( newItemId ,
tokenURI );
return
newItemId ;
}
function
mintNFTWithRoyalty ( address
recipient ,
string
memory
tokenURI ,
address
royaltyReceiver ,
uint96
```

feeNumerator)

public

onlyOwner

returns

(uint256)

{

uint256

tokenId

=

mintNFT (recipient ,

tokenURI);

_setTokenRoyalty (tokenId ,

royaltyReceiver ,

feeNumerator);

return

tokenId ;

} }

Deploy the Smart Contracts Locally¶

In order to deploy our smart contracts, we'll need to modifymigrations/1_deploy_contracts.js like so:

const

RoyalPets

=

artifacts . require ("Royalpets"); module . exports

=

function

(deployer)

{

deployer . deploy (RoyalPets); }; Next, let's get a local Ganache instance up. There are a variety of ways to do so: through the VS Code extension, Ganache CLI, and the Ganche graphical user interface. Each has its own advantages, and you can check out v7's coolest featureshere .

In this tutorial, we'll be using the GUI. Open it up, create a workspace, and hit save (feel free to add your project to use some of the nifty features from the Ganache UI)!

This creates a running Ganache instance at HTTP://127.0.0.1:7545.

Next, uncomment thedevelopment network in yourtruffle-config.js and modify the port number to 7545 to match.

development :

{

host :

"127.0.0.1" ,

// Localhost (default: none)

port :

7545 ,

// Standard Ethereum port (default: none)

network_id :

"*" ,

// Any network (default: none) } Now, simply runtruffle migrate , which defaults to thedevelopment network, to deploy! You can also deploy from the VS Code extension as well. Then, you can see your built contracts inbuild/contracts , from the VS Code extension, or in your Ganache UI!

Test Your Smart Contract¶

If you want to test your smart contract commands on the fly without writing a full test, you can do so throughtruffle develop ortruffle console . Read more about ithere .

For the purposes of this tutorial, we'll just go ahead and write a Javascript test. Note that with Truffle, you have the option of writing tests in Javascript, Typescript, or Solidity.

const

RoyalPets

=

artifacts . require ("RoyalPets"); contract ("RoyalPets" ,

function

(accounts)

{

it ("should support the ERC721 and ERC2198 standards" ,

async

()

=>

{

const

royalPetsInstance

=

await

RoyalPets . deployed ();

const

ERC721InterfaceId

=

"0x80ac58cd" ;

const

ERC2981InterfaceId

=

"0x2a55205a" ;

```
var
isERC721
=
await
royalPetsInstance . supportsInterface ( ERC721InterfaceId );
var
isER2981
=
await
royalPetsInstance . supportsInterface ( ERC2981InterfaceId );
assert . equal ( isERC721 ,
true ,
"RoyalPets is not an ERC721" );
assert . equal ( isER2981 ,
true ,
"RoyalPets is not an ERC2981" );
});
it ( "should return the correct royalty info when specified and burned" ,
async
()
=>
{
const
royalPetsInstance
=
await
RoyalPets . deployed ();
await
royalPetsInstance . mintNFT ( accounts [ 0 ],
"fakeURI" );
// Override royalty for this token to be 10% and paid to a different account
await
royalPetsInstance . mintNFTWithRoyalty ( accounts [ 0 ],
"fakeURI" ,
accounts [ 1 ],
1000 );
const
```

```
defaultRoyaltyInfo
=
await
royalPetsInstance . royaltyInfo . call ( 1 ,
1000 );
var
tokenRoyaltyInfo
=
await
royalPetsInstance . royaltyInfo . call ( 2 ,
1000 );
const
owner
=
await
royalPetsInstance . owner . call ();
assert . equal ( defaultRoyaltyInfo [ 0 ],
owner ,
"Default receiver is not the owner" );
// Default royalty percentage taken should be 1%.
assert . equal ( defaultRoyaltyInfo [ 1 ]. toNumber (),
10 ,
"Royalty fee is not 10" );
assert . equal ( tokenRoyaltyInfo [ 0 ],
accounts [ 1 ],
"Royalty receiver is not a different account" );
// Default royalty percentage taken should be 1%.
assert . equal ( tokenRoyaltyInfo [ 1 ]. toNumber (),
100 ,
"Royalty fee is not 100" );
// Royalty info should be set back to default when NFT is burned
await
royalPetsInstance . burnNFT ( 2 );
tokenRoyaltyInfo
=
await
royalPetsInstance . royaltyInfo . call ( 2 ,
```

1000);

assert . equal (tokenRoyaltyInfo [0],

owner ,

"Royalty receiver has not been set back to default");

assert . equal (tokenRoyaltyInfo [1]. toNumber (),

10 ,

"Royalty has not been set back to default");

}); }); And, finally, just calltruffle test !

Contract: RoyalPets ✔ should support the ERC721 and ERC2198 standards (67ms)

✔ should return

the correct royalty info when specified and burned (1077ms) 2

passing (1s)

Mint an NFT and View it in Your Mobile Wallet or OpenSea

If you want to mint an NFT for yourself and view it in your mobile MetaMask wallet, you'll need to deploy your contract to a public testnet or mainnet. To do so, you'll need to grab your Infura project API from your Infura project and your MetaMask wallet secret key. At the root of your folder, add a.env file, in which we'll put in that information.

WARNING: DO NOT PUBLICIZE OR COMMIT THIS FILE. We recommend adding.env to a.gitignore file.

MNEMONIC

"YOUR SECRET KEY" INFURA_API_KEY = "YOUR INFURA_API_KEY" Then, at the top oftruffle-config.js , add this code to get retrieve that information:

require ('dotenv'). config (); const

mnemonic

=

process . env ["MNEMONIC"]; const

infuraApiKey

=

process . env ["INFURA_API_KEY"]; const

HDWalletProvider

=

require ('@truffle/hdwallet-provider'); And finally, add the Goerli network to thenetworks list undermodule.exports :

goerli :

{

provider :

()

=>

new

HDWalletProvider (mnemonic ,

https://goerli.infura.io/v3/ { infuraApiKey }),

network_id :

5 ,

// Goerli's network id

chain_id :

5 ,

// Goerli's chain id

gas :

5500000 ,

// Gas limit used for deploys.

confirmations :

2 ,

// # of confirmations to wait between deployments. (default: 0)

timeoutBlocks :

200 ,

// # of blocks before a deployment times out (minimum/default: 50)

skipDryRun :

true

// Skip dry run before migrations? (default: false for public nets) } Your finaltruffle-config.js should look something like this:

require ('dotenv'). config (); const

mnemonic

=

process . env ["MNEMONIC"]; const

infuraApiKey

=

process . env ["INFURA_API_KEY"]; const

HDWalletProvider

=

require ('@truffle/hdwallet-provider'); module . exports

=

{

networks :

{

development :

{

host :

"127.0.0.1" ,

// Localhost (default: none)

```
port :
7545 ,
// Standard Ethereum port (default: none)
network_id :
"*" ,
// Any network (default: none)
},
goerli :
{
provider :
()
=>
new
HDWalletProvider ( mnemonic ,
https://goerli.infura.io/v3/ { infuraApiKey } ),
network_id :
5 ,
// Goerli's network id
chain_id :
5 ,
// Goerli's chain id
gas :
5500000 ,
// Gas limit used for deploys.
confirmations :
2 ,
// # of confirmations to wait between deployments. (default: 0)
timeoutBlocks :
200 ,
// # of blocks before a deployment times out (minimum/default: 50)
skipDryRun :
true
// Skip dry run before migrations? (default: false for public nets)
}
},
// Set default mocha options here, use special reporters, etc.
mocha :
```

{

// timeout: 100000

},

// Configure your compilers

compilers :

{

solc :

{

version :

"0.8.15" ,

// Fetch exact version from solc-bin (default: truffle's version)

}

}, }; Then, we'll need to install the dev dependencies fordotenv and@truffle/hdwallet-provider . Lastly, runtruffle migrate --network goerli to deploy!

npm i --save-dev dotenv npm i --save-dev @truffle/hdwallet-provider truffle migrate --network goerli Then, to quickly interact with the goerli network, we can usetruffle console --network goerli , and call the appropriate contract functions. We've already pinned some metadata to IPFS for you to use as your tokenURI:ipfs://bafybeiffapvkruv2vwtomswqzxiaxdgm2dflet2cxmh6t4ixrgaezumbw4 . It should look a bit like this:

truffle migrate --network goerli truffle(goerli)

const contract

=

await RoyalPets.deployed() undefined truffle(goerli)

await contract.mintNFT("YOUR ADDRESS" , "ipfs://bafybeiffapvkruv2vwtomswqzxiaxdgm2dflet2cxmh6t4ixrgaezumbw4") If you want to populate your own metadata, there are a variety of ways to do so - with either Truffle or Infura. Check out the guides here: -truffle preserve -infura IPFS

To view your NFT on your mobile wallet, open up MetaMask mobile, switch to the Goerli network, and open the NFTs tab! To view on OpenSea, you'll have to deploy to mainnet or Polygon. Otherwise, if you deploy your contract torinkeby , you can view it onhttps://testnets.opensea.io/ . To be aware thatrinkeby will be deprecated afterthe merge .

If you don't want to monitor your transactions in an Infura project, you can also deploy viaTruffle Dashboard , which allows you to deploy and sign transactions via MetaMask - thus never revealing your private key! To do so, simply run:

truffle dashboard truffle migrate --network dashboard truffle console --network dashboard

Future Extensions¶

And there you have it! You've written an NFT smart contract that can be queried for royalty information. Look out for a more in-depth guide for uploading your metadata to IPFS! For a more a detailed walkthrough of the code, be sure to watch the livestream onYouTube . In future editions of Web3 Unleashed, expect to see how we can make our basic ERC-721s rentable by implementing ERC-4907 as well as creating a NFT rental marketplace that uses the various NFT standards that exist!

Some additional extensions you might consider is overriding the wayroyaltyInfo is returned. Gemini has a cool blog detailing some such as decaying royalties, multisig royalties, and stepped royaltieshere . Let us know if you try any of them out!

If you want to talk about this content, make suggestions for what you'd like to see or ask questions about the series, start a discussionhere . If you want to show off what you built or just hang with the Unleashed community in general, join ourDiscord ! Lastly, don't forget to follow us onTwitter for the latest updates on all things Truffle.