Web3 Unleashed: Write a Rentable NFT Smart Contract¶

Written byEmily Lin and Leandro Faria

Last updated 11/15/2022

Overview¶

In this guide, we'll be covering what the ERC-4907 rentable NFT standard is and how we can implement one using Truffle!

Watch our livestream recording with Jesse Luong fromDouble Protocol , the creators of theERC-4907 standard onYouTube for a more in-depth explanation and exploration into the standard's impact on GameFi and the metaverse!

What is the ERC-4907?¶

NFT renting has become a growing use case for utility based NFTs - for example, virtual land in the metaverse or in-game NFT assets. In thefirst episode of Web3 Unleashed, we learned that ERCs are application level standards that establish a shared interface for contracts and dapps to reliably interact with each other. In this case, ERC-4907 standardizes the way NFT rentals happen by separating the concept ofuser andowner . This allows us to identify permissioned roles on the NFT. That is, auser has the ability to use the NFT, but does not have the permission to sell it. In addition, anexpires function is introduced, so that theuser only has temporary access to use the NFT.

What's in an ERC-4907?¶

The interface is specified as follows:

interface

IERC4907

{

// Logged when the user of a NFT is changed or expires is changed

/// @notice Emitted when theuser of an NFT or theexpires of theuser is changed

/// The zero address for user indicates that there is no user address

event

UpdateUser (uint256

indexed

tokenId ,

address

indexed

user ,

uint64

expires);

/// @notice set the user and expires of a NFT

/// @dev The zero address indicates there is no user

/// Throws iftokenId is not valid NFT

/// @param user The new user of the NFT

/// @param expires UNIX timestamp, The new user could use the NFT before expires

function

setUser (uint256

tokenId ,

address

user ,

uint64

expires)

external ;

/// @notice Get the user address of an NFT

/// @dev The zero address indicates that there is no user or the user is expired

/// @param tokenId The NFT to get the user address for

/// @return The user address for this NFT

function

userOf (uint256

tokenId)

external

view

returns (address);

/// @notice Get the user expires of an NFT

/// @dev The zero value indicates that there is no user

/// @param tokenId The NFT to get the user expires for

/// @return The user expires for this NFT

function

userExpires (uint256

tokenId)

external

view

returns (uint256); } Additionally: - TheuserOf(uint256 tokenId) function MAY be implemented aspure orview . - TheuserExpires(uint256 tokenId) function MAY be implemented aspure orview . - ThesetUser(uint256 tokenId, address user, uint64 expires) function MAY be implemented aspublic orexternal . - TheUpdateUser event MUST be emitted when a user address is changed or the user expires is changed. - ThesupportsInterface method MUST return true when called with0xad092b5c .

Let's Write an ERC-4907

Let's get started writing a rentable NFT! You can find the completed code here . 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 itrentable-nft , 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 calledrentable-nft .

truffle init rentable-nftcd

rentable-nft truffle create contract RentablePets truffle create contract IERC4907 truffle create contract ERC4907 truffle create test

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

rentable-nft ├—— contracts | ├—— ERC4907.sol | ├—— IERC4907.sol | └—— RentablePets.sol ├—— migrations | └—— 1_deploy_contracts.js ├—— test | └—— test_rentable_pets.js └—— truffle-config.js

Write the ERC-4907 Interface¶

Now, let's add the interface functions defined in theEIP . To do this, go toIERC4907.sol and changecontract tointerface . Then, we'll just copy and paste what was specified on the EIP! It should look like this:

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; interface

IERC4907

{

// Logged when the user of a NFT is changed or expires is changed

/// @notice Emitted when theuser of an NFT or theexpires of theuser is changed

/// The zero address for user indicates that there is no user address

event

UpdateUser (uint256

indexed

tokenId ,

address

indexed

user ,

uint64

expires);

/// @notice set the user and expires of a NFT

/// @dev The zero address indicates there is no user

/// Throws if tokenId is not valid NFT

/// @param user The new user of the NFT

/// @param expires UNIX timestamp, The new user could use the NFT before expires

function

setUser (uint256

tokenId ,

address

user ,

uint64

expires)

external ;

/// @notice Get the user address of an NFT

/// @dev The zero address indicates that there is no user or the user is expired

/// @param tokenId The NFT to get the user address for

/// @return The user address for this NFT

function

userOf (uint256

tokenId)

external

view

returns (address);

/// @notice Get the user expires of an NFT

/// @dev The zero value indicates that there is no user

/// @param tokenId The NFT to get the user expires for

/// @return The user expires for this NFT

function

userExpires (uint256

tokenId)

external

view

returns (uint256); } Once you've created this file you shouldn't need to touch it again.

Write the ERC-4907 Smart Contract¶

Now, let's write anERC-4907 smart contract that extends OpenZeppelin'sERC-721URIStorage contract. First, install OpenZeppelin's contracts:

npm i @openzeppelin/contracts@4.8.0 The basics of an ERC-721 are covered in this Infura blog . We choose to useERC721URIStorage so that we don't have to use a static metadata file to populate thetokenURI . To do this, import the interface we created and OpenZeppelin'sERC721URIStorage implementation and have ourERC4907 smart contract inherit their properties as follows:

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; import

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

"./IERC4907.sol" ; contract

ERC4907

is

ERC721URIStorage ,

IERC4907

{

constructor ()

public

{

} } Then, we'll modify the constructor to take in the NFT collection name and symbol when the contract is deployed.

contract

ERC4907

is

ERC721 ,

IERC4907

{

constructor (string

memory

_name ,

string

memory

_symbol)

ERC721 (_name ,

_symbol){

} } Before we start implementing the functions defined inIERC4907 , let's set up two state variablesUserInfo and_users to help define and store the concept ofuser .

contract

ERC4907

is

ERC721URIStorage ,

IERC4907

{

struct

UserInfo

{

address

user ;

// address of user role

uint64

expires ;

// unix timestamp, user expires

}

mapping (uint256

=>

UserInfo)

internal

_users ; * UserInfo * stores the user's address and the rental expiration date * _users * maps thetokenId * of the relevant NFT to the appropriateuser * (rentee)

Finally, let's get started on implementing the interface functions!

setUser¶

This function can only be called by theowner of the NFT. It allows the owner to specify who will be the rentee of the NFT. The user now has the NFT in their wallet, but cannot perform any actions on it such as burn or transfer. Add this function to yourERC4907.sol file:

/// @notice set the user and expires of a NFT /// @dev The zero address indicates there is no user /// Throws if tokenId is not valid NFT /// @param user The new user of the NFT /// @param expires UNIX timestamp, The new user could use the NFT before expires function

setUser (uint256

tokenId ,

address

user ,

uint64

expires)

public

virtual

override

{

require (_isApprovedOrOwner (msg . sender ,

tokenId), "ERC721: transfer caller is not owner nor approved");

UserInfo

storage

info

=

_users [tokenId];

info . user

=

user ;

info . expires

=

expires ;

emit

UpdateUser (tokenId ,

user ,

expires); } This function will update theUserInfo struct with theaddress of the rentee and the block timestamp that the renting period willexpires . We use the inherited function_isApprovedOrOwner fromERC721 to indicate only the owner has the ability to decide who can user the NFT. Lastly, we will emit anUpdateUser event defined inIERC4907 to communicate relevant information when setting a new user.

userOf¶

Next, we want to be able to identify who the current user of an NFT is. AdduserOf to your contract:

/// @notice Get the user address of an NFT /// @dev The zero address indicates that there is no user or the user is expired /// @param tokenId The NFT to get the user address for /// @return The user address for this NFT function

userOf (uint256

tokenId)

public

view

virtual

override

returns

(address) {

if

(uint256 (_users [tokenId]. expires)

=

block . timestamp)

{

return

_users [tokenId]. user ;

}

else

{

return

address (0);

} } This function takes thetokenId as an argument and will return the useraddress if that token is still being rented. Otherwise, the zero address indicates that the NFT is not being rented.

userExpires¶

Add theuserExpires function so that dapps can retrieve expiration date information for a specific NFT:

/// @notice Get the user expires of an NFT /// @dev The zero value indicates that there is no user /// @param tokenId The NFT to get the user expires for /// @return The user expires for this NFT function

userExpires (uint256

tokenId)

public

view

virtual

override

returns (uint256){

return

_users [tokenId]. expires ; } IftokenId does not exist, then aUserInfo with default values will be returned. In this case, the default for theuser address will beaddress(0) , andexpires , which is anuint64 , will be0 .

supportsInterface¶

In order for a dapp to know whether or not our NFT is rentable, it needs to be able to check for theinterfaceId ! To do so, override thesupportsInterface function as defined in theEIP-165 standard .

/// @dev See {IERC165-supportsInterface}. function

supportsInterface (bytes4

interfaceId)

public

view

virtual

override

returns

(bool) {

return

interfaceId

==

type (IERC4907). interfaceId

||

super . supportsInterface (interfaceId); }

_beforeTokenTransfer¶

This is the final function we will implement! When the token is transferred (i.e., the owner changes) or burned, we want to remove the rental information as well. Note that this behavior is inherited from OpenZeppelin'sERC721 implementation. We will override_beforeTokenTransfer fromERC721 to add in this functionality:

function

_beforeTokenTransfer (

address

from ,

address

to ,

uint256

tokenId ,

uint256

batchSize)

internal

virtual

override

{

super . _beforeTokenTransfer (from ,

to ,

tokenId ,

batchSize);

if

(from

!=

to

&&

_users [tokenId]. user

!=

address (0))

{

delete

_users [tokenId];

emit

UpdateUser (tokenId ,

address (0),

0);

} } In order to delete theUserInfo from the mapping, we want to make sure there was actually a transfer of ownership and there was UserInfo on it in the first place. Once verified, we can delete and emit an event that the UserInfo was updated!

Note that it is up to you, the contract writer, to decide if this is how you expect token transfers and burns to behave. You might choose to ignore this and say that rentees maintain their user status even when ownership changes!

Now, your final contract should look like this:

// SPDX-License-Identifier: MIT pragma

solidity

^ 0.8.0 ; import

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

"./IERC4907.sol" ; contract

ERC4907

is

ERC721URIStorage ,

IERC4907

{

struct

UserInfo

{

address

user ;

// address of user role

uint64

expires ;

// unix timestamp, user expires

}

mapping (uint256

=>

UserInfo)

internal

_users ;

constructor (string

memory

name_ ,

string

memory

symbol_)

ERC721 (name_ ,

symbol_)

{}

/// @notice set the user and expires of a NFT

```
/// @dev The zero address indicates there is no user
/// Throws if tokenId is not valid NFT
/// @param user The new user of the NFT
/// @param expires UNIX timestamp, The new user could use the NFT before expires
function
setUser (
uint256
tokenId ,
address
user ,
uint64
expires
)
public
virtual
override
{
require (
_isApprovedOrOwner ( msg . sender ,
tokenId ),
"ERC721: transfer caller is not owner nor approved"
);
UserInfo
storage
info
=
_users [ tokenId ];
info . user
=
user ;
info . expires
=
expires ;
emit
UpdateUser ( tokenId ,
user ,
expires );
```

```
}
/// @notice Get the user address of an NFT
/// @dev The zero address indicates that there is no user or the user is expired
/// @param tokenId The NFT to get the user address for
/// @return The user address for this NFT
function
userOf ( uint256
tokenId )
public
view
virtual
override
returns
( address )
{
if
( uint256 ( _users [ tokenId ]. expires )
        =
block . timestamp )
{
return
_users [ tokenId ]. user ;
}
else
{
return
address ( 0 );
}
}
/// @notice Get the user expires of an NFT
/// @dev The zero value indicates that there is no user
/// @param tokenId The NFT to get the user expires for
/// @return The user expires for this NFT
function
userExpires ( uint256
tokenId )
public
```

```
view
virtual
override
returns
( uint256 )
{
return
_users [ tokenId ]. expires ;
}
/// @dev See {IERC165-supportsInterface}.
function
supportsInterface ( bytes4
interfaceId )
public
view
virtual
override
returns
( bool )
{
return
interfaceId
==
type ( IERC4907 ). interfaceId
||
super . supportsInterface ( interfaceId );
}
function
_beforeTokenTransfer (
address
from ,
address
to ,
uint256
tokenId
)
internal
```

virtual

override

{

super . _beforeTokenTransfer (from ,

to ,

tokenId);

if

(from

!=

to

&&

_users [tokenId]. user

!=

address (0))

{

delete

_users [tokenId];

emit

UpdateUser (tokenId ,

address (0),

0);

}

} }

Write the RentablePets Smart Contract¶

Finally, we can write an NFT that utilizes theERC4907 contract we just implemented. We are following the same NFT format as written in previous guides. You can look through those for a more in-depth explanation. We're exposing theburn function so that we can test it. Don't include this method if you don't want your NFT to be transferrable!

Your final contract should look like this:

// SPDX-License-Identifier: MIT pragma

solidity

= 0.4.22

< 0.9.0 ; import

"./ERC4907.sol" ; import

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

RentablePets

is

ERC4907

{

using

Counters

for

Counters . Counter ;

Counters . Counter

private

_tokenIds ;

constructor ()

ERC4907 ("RentablePets" ,

"RP")

{}

function

mint (string

memory

_tokenURI)

public

{

_tokenIds . increment ();

uint256

newTokenId

=

_tokenIds . current ();

_safeMint (msg . sender ,

newTokenId);

_setTokenURI (newTokenId ,

_tokenURI);

}

function

burn (uint256

tokenId)

public

{

_burn (tokenId);

} }

Start a Local Blockchain¶

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

const

RentablePets

=

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

=

function

(deployer)

{

deployer . deploy (RentablePets); }; 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) }

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.

We want to test the following functionality: 1. That RentablePets is an ERC721 and ERC4907 2. ThatsetUser cannot be called by someone other than the owner 3. ThatsetUser can be correctly called by the owner 4. Thatburn will properly deleteUserInfo

As part of testing, we'll want to make sure that events are properly emitted, as well as ourrequire statement failing correctly. OpenZeppelin has some really niftytest helpers that we'll be using. Download it:

npm install --save-dev @openzeppelin/test-helpers The complete test looks like this:

require ("@openzeppelin/test-helpers/configure")({

provider :

web3 . currentProvider ,

singletons :

{

abstraction :

"truffle" ,

}, }); const

{

constants ,

expectRevert ,

expectEvent

}

=

require ('@openzeppelin/test-helpers'); const

RentablePets

=

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

function

(accounts)

{

it ("should support the ERC721 and ERC4907 standards" ,

async

()

=>

{

const

rentablePetsInstance

=

await

RentablePets . deployed ();

const

ERC721InterfaceId

=

"0x80ac58cd" ;

const

ERC4907InterfaceId

=

"0xad092b5c" ;

var

isERC721

=

```
await
rentablePetsInstance . supportsInterface ( ERC721InterfaceId );
var
isER4907
=
await
rentablePetsInstance . supportsInterface ( ERC4907InterfaceId );
assert . equal ( isERC721 ,
true ,
"RentablePets is not an ERC721" );
assert . equal ( isER4907 ,
true ,
"RentablePets is not an ERC4907" );
});
it ( "should not set UserInfo if not the owner" ,
async
()
=>
{
const
rentablePetsInstance
=
await
RentablePets . deployed ();
const
expirationDatePast
=
1660252958 ;
// Aug 8 2022
await
rentablePetsInstance . mint ( "fakeURI" );
// Failed require in function
await
expectRevert ( rentablePetsInstance . setUser ( 1 ,
accounts [ 1 ],
expirationDatePast ,
{ from :
```

```
accounts [ 1 ]}),
"ERC721: transfer caller is not owner nor approved" );
// Assert no UserInfo for NFT
var
user
=
await
rentablePetsInstance . userOf . call ( 1 );
var
date
=
await
rentablePetsInstance . userExpires . call ( 1 );
assert . equal ( user ,
constants . ZERO_ADDRESS ,
"NFT user is not zero address" );
assert . equal ( date ,
0 ,
"NFT expiration date is not 0" );
});
it ( "should return the correct UserInfo" ,
async
()
=>
{
const
rentablePetsInstance
=
await
RentablePets . deployed ();
const
expirationDatePast
=
1660252958 ;
// Aug 8 2022
const
expirationDateFuture
```

=

4121727755 ;

// Aug 11 2100

await

rentablePetsInstance . mint ("fakeURI");

await

rentablePetsInstance . mint ("fakeURI");

// Set and get UserInfo

var

expiredTx

=

await

rentablePetsInstance . setUser (2 ,

accounts [1],

expirationDatePast)

var

unexpiredTx

=

await

rentablePetsInstance . setUser (3 ,

accounts [2],

expirationDateFuture)

var

expiredNFTUser

=

await

rentablePetsInstance . userOf . call (2);

var

expiredNFTDate

=

await

rentablePetsInstance . userExpires . call (2);

var

unexpireNFTUser

=

await

rentablePetsInstance . userOf . call (3);

```
var
unexpiredNFTDate
=
await
rentablePetsInstance . userExpires . call ( 3 );
// Assert UserInfo and event transmission
assert . equal ( expiredNFTUser ,
constants . ZERO_ADDRESS ,
"Expired NFT has wrong user" );
assert . equal ( expiredNFTDate ,
expirationDatePast ,
"Expired NFT has wrong expiration date" );
expectEvent ( expiredTx ,
"UpdateUser" ,
{
tokenId :
"2" ,
user :
accounts [ 1 ],
expires :
expirationDatePast . toString ()});
assert . equal ( unexpireNFTUser ,
accounts [ 2 ],
"Expired NFT has wrong user" );
assert . equal ( unexpiredNFTDate ,
expirationDateFuture ,
"Expired NFT has wrong expiration date" );
expectEvent ( unexpiredTx ,
"UpdateUser" ,
{
tokenId :
"3" ,
user :
accounts [ 2 ],
expires :
expirationDateFuture . toString ()});
// Burn NFT
```

unexpiredTx

=

await

rentablePetsInstance . burn (3);

// Assert UserInfo was deleted

unexpireNFTUser

=

await

rentablePetsInstance . userOf . call (3);

unexpiredNFTDate

=

await

rentablePetsInstance . userExpires . call (3);

assert . equal (unexpireNFTUser ,

constants . ZERO_ADDRESS ,

"NFT user is not zero address");

assert . equal (unexpiredNFTDate ,

0 ,

"NFT expiration date is not 0");

expectEvent (unexpiredTx ,

"UpdateUser" ,

{

tokenId :

"3" ,

user :

constants . ZERO_ADDRESS ,

expires :

"0" });

}); }); There's one special thing to call out here: To test thatsetUser fails whenmsg.sender is notowner , we can fake the sender by adding the extrafrom param:

rentablePetsInstance . setUser (1 ,

accounts [1],

expirationDatePast ,

{ from :

accounts [1]}) If you run into issues testing, using theTruffle debugger is really helpful!

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 RentablePets.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 a rentable NFT contract! 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, get excited to integrate this into a full-stack DApp. That is, a NFT rental marketplace that will use both the ERC-4907 rentable standard and ERC-2981 royalty standard

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.