Web3 Unleashed: Decentralized social media with Lens¶

Written byEmily Lin

Last updated 1/31/2023

Overview¶

Web3 is revolutionizing the social media landscape. In this guide, we'll cover how to use the Truffle Lens box to start building your social media dapp. We'll walk through what's in it and provide an example of how you can customize the Lens Protocol by creating a Follow Module.

Watch the livestream onYouTube to hear from Nader Dabit about how Lens Protocol is onboarding the next million of web3 users.

The Lens box code liveshere .

Download System Requirements¶

You'll need to install:

- Node.js
- , v14 or higher
- truffle
- ganache 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 .

Register for a MetaMask wallet¶

To interact with your DApp in the browser, you'll need a MetaMask wallet. You can download it and create onhere .

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 a site that links to different Goerli and Sepolia ETH faucets.

Unbox the Truffle box¶

First, let's examine the contents of the Truffle Lens box. Start off by unboxing it:

truffle unbox lens In this box, we have two folders:lens-app andtruffle . Let's dive into what's in each folder and how you might edit the contents to create your own social dapp!

lens-app

: your frontend code¶

You can build on Lens without writing any smart contracts because they've provided a robust API that will interact with their contracts for you. You can find the documentation for the APIhere .

lens-app contains frontend code that leverages Next.js and Tailwind CSS to build a app that:

1. Prompts the user to connect their wallet
2. Displays the top profiles on Lens
3. Allows the user to click into and see the posts by the top profiles on Lens

First, let's install the dependencies:

cd

lens-app npm i Let's dive into the important pieces of code that demonstrate how to leverage the Lens API. This assumes you already have understanding of Next.js and frontend development.

lens-app/api.js

¶

In order to interact with Lens, we first need to create the api. To do this, you'll need to construct GraphQL queries. We're doing so withApollo client .

The first thing we do in this file is create our Apollo client. While reading data from the Lens API is as simple as sending a GraphQL query, we need to either be authenticated or write a transaction directly to the Lens smart contracts to make any state change, like following, unfollowing, creating a post, and creating a mirror. Lines 1-23 demonstrate how to create an authenticated Apollo client:

import

{

ApolloClient ,

InMemoryCache ,

gql ,

createHttpLink

}

from

'@apollo/client' import

{

setContext

}

from

'@apollo/client/link/context' ; const

API_URL

=

'https://api.lens.dev' const

authLink

=

setContext ((_ ,

{

headers

})

=>

{

const

token

=

window . localStorage . getItem ('your-storage-key')

return

{

headers :

{

... headers ,

authorization :

token

?

Bearer { token }

:

"" ,

}

} }) const

httpLink

=

createHttpLink ({

uri :

API_URL }) export

const

client

=

new

ApolloClient ({

link :

authLink . concat (httpLink),

cache :

new

InMemoryCache () }) Do note that we are getting our authenticated token fromlocalStorage , which we write to inindex.tsx . You can read more about it in the section below.

The remaining code in this file represent different GraphQL queries that get the information we want from Lens. Thechallenge andauthenticate queries are specifically used for authentication, whileexploreProfiles ,getProfile , andgetPublications are for reading data from Lens.

lens-app/pages/index.tsx

¶

This is the home page of our dapp. It requires the user to be connected and authenticated before rendering all the Lens profile information. You can read more about the login processhere .

To do the authentication, the important functions to highlight are:

1. checkConnection
2. async
3. function
4. checkConnection
5. ()
6. {
7. const
8. provider
9. =
10. new
11. ethers
12. .
13. providers
14. .
15. Web3Provider
16. (
17. window
18. .
19. ethereum
20.)
21. const
22. accounts
23. =
24. await
25. provider
26. .
27. listAccounts
28. ()
29. if
30. (
31. accounts
32. .
33. length
34.)
35. {
36. setAddress
37. (
38. accounts
39. [
40. 0
41.])
42. }
43. }
44. This function checks to see if the user has already connected their wallet when the app loads and saves the address of the connected account.
45. connect
46. async
47. function
48. connect
49. ()
50. {
51. / *this allows the user to connect their wallet*/
52. const
53. account
54. =
55. await
56. window
57. .
58. ethereum
59. .
60. send
61. (
62. 'eth_requestAccounts'
63.)
64. if
65. (
66. account
67. .

68. result
69. .
70. length
71.)
72. {
73. setAddress
74. (
75. account
76. .
77. result
78. [
79. 0
80.])
81. }
82. }
83. If the user has not yet connected their account, this function will allow them to do so by using the MetaMask Provider APIawait window.ethereum.send('eth_requestAccounts')
84. .
85. login
86. async
87. function
88. login
89. ()
90. {
91. try
92. {
93. / *first request the challenge from the API server*/
94. const
95. challengeInfo
96. =
97. await
98. client
99. .
100. query
101. ({
102. query
103. :
104. challenge
105. ,
106. variables
107. :
108. {
109. address
110. }
111. })
112. const
113. provider
114. =
115. new
116. ethers
117. .
118. providers
119. .
120. Web3Provider
121. (
122. window
123. .
124. ethereum
125.);
126. const
127. signer
128. =
129. provider
130. .
131. getSigner
132. ()
133. / *ask the user to sign a message with the challenge info returned from the server*
134. const

```
signature
=
await
signer
.
signMessage
(
challengeInfo
.
data
.
challenge
.
text
)
/ authenticate the user/
const
authData
=
await
client
.
mutate
({
mutation
:
authenticate
,
variables
:
{
address
,
signature
}
})
/ if user authentication is successful, you will receive an accessToken and refreshToken/
const
{
data
:
{
authenticate
:
{
accessToken
}}}
=
authData
console
.
log
({
accessToken
})
setToken
(
accessToken
)
window
.
localStorage
.
setItem
(
'your-storage-key'
,
accessToken
```

)
}
catch
(
err
)
{
console
.
log
(
'Error signing in: '
,
err
)
}
}
Finally, to get our token, we need to issue a challenge and get the user to sign it using their wallet. To do so, we'll be using thechallenge
andauthenticate
queries we created inapi.js
. Once we get the token, we save it tolocalStorage
. Note that we callsetToken(window.localStorage.getItem('your-storage-key'))
in theuseEffect
so that we don't have to re-authenticate every time we refresh the page.
Once the user is logged in, we display the top Lens profiles! Getting that information is as simple as calling theexploreProfiles
query we defined inapi.js
:
async
function
fetchProfiles
()
{
try
{
/ *fetch profiles from Lens API* /
let
response
=
await
client
.
query
({
query
:
exploreProfiles
})
/ *loop over profiles, create properly formatted ipfs image links*/
let
profileData
=
await
Promise
.
all
(
response
.
data
.
exploreProfiles
.
items
.
map
(

```
async
profileInfo
=>
{
let
profile
=
{
...
profileInfo
}
let
picture
=
profile
.
picture
if
(
picture
&&
picture
.
original
&&
picture
.
original
.
url
)
{
if
(
picture
.
original
.
url
.
startsWith
(
'ipfs://'
))
{
let
result
=
picture
.
original
.
url
.
substring
(
7
,
picture
.
original
.
url
.
length
)
profile
.
```

```
avatarUrl
=
`http://lens.infura-ipfs.io/ipfs/
{
result
}
`
}
else
{
profile
.
avatarUrl
=
picture
.
original
.
url
}
}
return
profile
}))
/ update the local state with the profiles array/
setProfiles
(
profileData
)
}
catch
(
err
)
{
console
.
log
({
err
})
}
}
```

lens-app/pages/profile/[handle].js

¶

The last hook-in to the Lens API is in[handle].js . If you notice on line 114 inindex.tsx , you can navigate to a detailed view of the user's profile. This will simply direct you to[handle].js , where we format the data queried from the Lens API:

const

returnedProfile

=

await

client . query ({

query :

getProfile ,

variables :

{

handle

} }) const

pubs

=

await

client . query ({

query :

getPublications ,

variables :

{

id :

profileData . id ,

limit :

50

} })

Running the dapp¶

To see this code in action, simply callnpm run dev . You can use this as a launching off point for more complex social dapps. If you don't need to write any smart contracts, you can just delete thetruffle folder.

truffle

: your smart contract code¶

Thetruffle folder contains the set up for if you want to buildLens modules to customize Lens' capabilities. For example, if you wanted to change the comment mechanism such that only NFT holders can comment, you can do that by writing smart contracts to create areference module . Lens will then call into that module at pre-determined points to execute your custom functionality!

Before we dive into creating our own module, let's go over what's in the box so far.

truffle/contracts

¶

This folder contains all the Lens protocol contracts. In order to create a module, we'll be adding a smart contract here undertruffle/contracts/core/modules .

truffle/migrations/1_deploy_lens_protocol.js

¶

This file deploys all the existing Lens Protocol contracts. There are some key pieces to highlight that are more complex than simply deploying individual contracts.

First off, we specify a few important addresses to take into account:

const

deployerAddress

=

accounts [0]; const

governanceAddress

=

accounts [1]; const

treasuryAddress

=

accounts [2]; const

proxyAdminAddress

=

deployerAddress ; const

profileCreatorAddress

=

deployerAddress ; Lens Protocol contracts are upgradeable contracts, which you can learn more about in our 3rd episode about upgradeble contracts with OpenZeppelin. Because upgradeable contracts are proxy contracts, we must provide an admin, who has the authority to upgrade the contracts should the need arise:

await

deployer . deploy (TransparentUpgradeableProxy ,

lensHubImpl . address ,

proxyAdminAddress ,

data ,

{

nonce :

deployerNonce ++

}); Moving forward, we only want to interact with the proxy address andnot theLensHub implementation contract. You can see us create our contract abstraction based off of the proxy in lines 114-116:

let

proxy

=

await

TransparentUpgradeableProxy . deployed (); let

lensHub

=

await

LensHub . at (proxy . address); Additionally, you'll note that we pass in the parameterdata to our proxy deployment, defined as follows:

let

data

=

await

web3 . eth . abi . encodeFunctionCall ({

"inputs" :

[

{

"internalType" :

"string" ,

"name" :

"name" ,

"type" :

"string"

},

{

"internalType" :

"string" ,

"name" :

"symbol" ,

"type" :

"string"

},

{

"internalType" :

"address" ,

"name" :

"newGovernance" ,

"type" :

"address"

}

],

"name" :

"initialize" ,

"outputs" :

[],

"stateMutability" :

"nonpayable" ,

"type" :

"function" },

[LENS_HUB_NFT_NAME ,

LENS_HUB_NFT_SYMBOL ,

governanceAddress]); We pass ingovernanceAddress , which is the only address that can call certainLensHub methods. In order to call a function such thatmsg.sender isgovernanceAddress , you can pass it in by modifying thefrom property like so:

await

lensHub . whitelistCollectModule (feeCollectModule . address ,

true ,

{

nonce :

governanceNonce ++ ,

from :

governanceAddress

}) We'll need to start a local test chain using Ganache to start interacting with this protocol. In addition to deploying each of our contracts, lines 207 to 245 in the migrations file whitelists our module smart contracts so that Lens can call into them.

What's also interesting about this deployment is linking library contracts:

console . log ('\n\t-- Deploying Hub Implementation --'); await

LensHub . link (hubLibs); await

deployer . deploy (LensHub ,

followNFTImplAddress ,

collectNFTImplAddress ,

{

nonce :

deployerNonce ++ ,

gas :

25000000

}); let

lensHubImpl

=

await

LensHub . deployed (); You cannot deploy contracts that are greater than 24.77 kib in size. In order to get around this restriction, there are two things we do:

1. Extracting functionality out into libraries
2. Optimizing contract compilation in ourtruffle-config.js
3. // Configure your compilers
4. compilers
5. :
6. {
7. solc
8. :
9. {
10. version
11. :
12. "0.8.10"
13. ,
14. // Fetch exact version from solc-bin (default: truffle's version)
15. // docker: true, // Use "0.5.1" you've installed locally with docker (default: false)
16. settings
17. :
18. {
19. // See the solidity docs for advice about optimization and evmVersion
20. optimizer
21. :
22. {

23. enabled
24. :
25. true
26. ,
27. runs
28. :
29. 200
30. },
31. // evmVersion: "byzantium"
32. }
33. }
34. },

You can read more about optimizershere . In short, the optimizer attempts to simplify complex code, with the tradeoff being deployment cost against execution cost. If this is off, this contract will fail to deploy!

The last piece that is interesting about this deployment is that we write all the relevant contract addresses to a file namedaddresses.json . This will be used later in our scripts when interacting with the Lens smart contracts.

truffle/scripts¶

This folder contains scripts that interact with the Lens protocol.

Inutils.js , we create some functions that help us easily retrieve common information. Note thatgetAddrs will read from the file we created in our migration script. Because it calls into the relative file path./addresses.json , you have to execute scripts from the roottruffle folder. Otherwise, the script will fail because it can't find./addresses.json .

You can only interact with it when it is unpaused, which you can do by callingtruffle exec scripts/unpause.js after you've deployed the Lens contracts.

Again, you'll note that we modify thefrom parameter in several contract calls.

1. We send from the governance address when calling functions inLensHub
2. contract that contains theonlyGov
3. modifier
4. await
5. lensHub
6. .
7. whitelistCollectModule
8. (
9. freeCollectModuleAddr
10. ,
11. true
12. ,
13. {
14. from
15. :
16. governance
17. });
18.
    1. We send from a user address when calling functions that perform user interactions, such as making a post
19. await
20. lensHub
21. .
22. post
23. (
24. inputStruct
25. ,
26. {
27. from
28. :
29. user
30. });

truffle/.env

andtruffle/truffle-config.js ¶

These two files define the networks that you can deploy the Lens contracts to. You can get the RPC URLs from your Infura

account and use the mnemonic from your MetaMask wallet. Be sure to never expose this information!

Running the protocol¶

In order to deploy our contracts locally, you need to spin up a local instance of Ganache:

ganache This will default to port 8545, which is designated as ourdevelopment network in ourtruffle-config.js . Then, to deploy:

truffle migrate If you want to deploy to other networks, you can run:

truffle migrate --network Because there are so many contracts, compilation will take some time. Do note that every time you run the migration, it will overwrite what contract addresses have been written toaddresses.json .

Build your own module¶

Now, we'll demonstrate how to create a custom module using the Truffle box. In this case, we'll only be working within thetruffle folder.

The completed code for the module liveshere .

Write the module smart contract¶

Let's customize the Lens follow functionality. Specifically, we want to specify that the user has to enter a password in order for them to follow a particular profile.

Start off by creating a smart contracttruffle/contracts/core/modules/follow/SecretCodeFollowModule.sol . Then, let's define our imports:

pragma

solidity

0.8.10 ; import

{ IFollowModule }

from

'../../../interfaces/IFollowModule.sol' ; import

{ ModuleBase }

from

'../ModuleBase.sol' ; import

{ FollowValidatorFollowModuleBase }

from

'./FollowValidatorFollowModuleBase.sol' ; 1. IFollowModule 2. defines the functions Lens will hook into to customize the follow behavior. To explain the functions it defines:* InitializeFollowModule() 3. * is called when a profile sets this module as its follow module. 4. * ProcessFollow() 5. * is called when a user attempts to follow a given profile with this module set as its follow module. 6. * FollowModuleTransferHook() 7. * is called when a FollowNFT associated with a profile that has this module set as its follow module is transferred 8. * ValidateFollow() 9. * whichis called to validate whether a follow is still valid 10. ModuleBase 11. exposes anonlyHub 12. modifier andHUB 13. address. 14. FollowValidatorFollowModuleBase 15. implementsisFollowing, which is one of the functions we need to define in the interface

Now, let's create a contract that inherits these imports:

contract

SecretCodeFollowModule

is

IFollowModule ,

FollowValidatorFollowModuleBase

{ } Then, let's define some variables we'll need. We'll create a custom error that indicates the wrong passcode was input and

a mapping that associates passwords with profiles.

error

PasscodeInvalid (); mapping (uint256

=>

uint256)

internal

_passcodeByProfile ; Then, addSecretCodeFollowModule 's constructor, which inherits fromModuleBase .

constructor (address

hub)

ModuleBase (hub)

{} Finally, we'll implement the interface functions:

function

initializeFollowModule (uint256

profileId ,

bytes

calldata

data)

external

override

onlyHub

returns

(bytes

memory) {

uint256

passcode

=

abi . decode (data ,

(uint256));

_passcodeByProfile [profileId]

=

passcode ;

return

data ; } function

processFollow (

address

follower ,

uint256

profileId ,

bytes

calldata

data)

external

view

override

{

uint256

passcode

=

abi . decode (data ,

(uint256));

if

(passcode

!=

_passcodeByProfile [profileId])

revert

PasscodeInvalid (); } function

followModuleTransferHook (

uint256

profileId ,

address

from ,

address

to ,

uint256

followNFTTokenId)

external

override

{} Note that we don't implement anything infollowModuleTransferHook because we don't need to use it! Your final smart contract code should look like this:

pragma

solidity

0.8.10 ; import

{ IFollowModule }

from

'../../../interfaces/IFollowModule.sol' ; import

```
{ ModuleBase }
from
'../ModuleBase.sol' ; import
{ FollowValidatorFollowModuleBase }
from
'./FollowValidatorFollowModuleBase.sol' ; contract
SecretCodeFollowModule
is
IFollowModule ,
FollowValidatorFollowModuleBase
{
error
PasscodeInvalid ();
mapping ( uint256
=>
uint256 )
internal
_passcodeByProfile ;
constructor ( address
hub )
ModuleBase ( hub )
{}
function
initializeFollowModule ( uint256
profileId ,
bytes
calldata
data )
external
override
onlyHub
returns
( bytes
memory )
{
uint256
passcode
```

```
=
abi . decode ( data ,
( uint256 ));
_passcodeByProfile [ profileId ]
=
passcode ;
return
data ;
}
function
processFollow (
address
follower ,
uint256
profileId ,
bytes
calldata
data
)
external
view
override
{
uint256
passcode
=
abi . decode ( data ,
( uint256 ));
if
( passcode
!=
_passcodeByProfile [ profileId ])
revert
PasscodeInvalid ();
}
function
followModuleTransferHook (
```

uint256

profileId ,

address

from ,

address

to ,

uint256

followNFTTokenId

)

external

override

{} }

Deploy your new contract

Let's create a new filetruffle/migrations/2_deploy_SecretCodeFollowModule.js . As in our previous migration file, we have to define ourLensHub contract based on the proxy address. Then, in order for us to use the new module, we have to whitelist it, calling the function from thegovernanceAddress .

const

SecretCodeFollowModule

=

artifacts . require ("SecretCodeFollowModule"); const

TransparentUpgradeableProxy

=

artifacts . require ("TransparentUpgradeableProxy"); const

LensHub

=

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

=

async

function

(deployer ,

networks ,

accounts)

{

const

governanceAddress

=

accounts [1];

const

proxy

=

await

TransparentUpgradeableProxy . deployed ();

const

lensHub

=

await

LensHub . at (proxy . address);

await

deployer . deploy (SecretCodeFollowModule ,

lensHub . address)

const

secretCodeFollowModule

=

await

SecretCodeFollowModule . deployed ();

await

lensHub . whitelistFollowModule (secretCodeFollowModule . address ,

true ,

{ from :

governanceAddress }); } Since we don't want to rerun the first migration, you can use the--f flag to specify exactly which migration file you want to execute:

truffle migrate --f 2

Write a script to test the new module¶

Now, let's write a script that will call on the new follow functionality. Create a filetruffle/scripts/secret_follow.js , and add this code:

const

{

defaultAbiCoder

}

=

require ('ethers/lib/utils'); const

{

getAddrs ,

initEnv ,

ProtocolState ,

ZERO_ADDRESS , }

=

require ('./helpers/utils'); const

LensHub

=

artifacts . require ("LensHub"); const

FollowNFT

=

artifacts . require ("FollowNFT"); const

SecretCodeFollowModule

=

artifacts . require ("SecretCodeFollowModule"); const

main

=

async

(cb)

=>

{

try

{

const

[governance ,

,

user]

=

await

initEnv (web3);

const

addrs

=

getAddrs ();

const

lensHub

=

await

LensHub . at (addrs ['lensHub proxy']);

await

lensHub . setState (ProtocolState . Unpaused ,

```
{ from :
governance });
await
lensHub . whitelistProfileCreator ( user ,
true ,
{ from :
governance });
// Will fail if you've already minted this profile
// const inputStruct = {
// to: user,
// handle: 'zer0dot',
// imageURI:
// 'https://ipfs.fleek.co/ipfs/ghostplantghostplantghostplantghostplantghostplantghostplan',
// followModule: ZERO_ADDRESS,
// followModuleInitData: [],
// followNFTURI:
// 'https://ipfs.fleek.co/ipfs/ghostplantghostplantghostplantghostplantghostplantghostplan',
// };
// await lensHub.createProfile(inputStruct, {from: user});
const
data
=
defaultAbiCoder . encode ([ 'uint256' ],
[ '42069' ]);
const
secretCodeFollowModule
=
await
SecretCodeFollowModule . deployed ();
await
lensHub . setFollowModule ( 1 ,
secretCodeFollowModule . address ,
data ,
{ from :
user });
try
{
```

```
await
lensHub . follow ([ 1 ],
[ badData ],
{ from :
user ,
gas :
"0xfffff" });
}
catch
( e )
{
console . log (Expected failure occurred! Error: { e } );
}
await
lensHub . follow ([ 1 ],
[ data ],
{ from :
user });
const
followNFTAddr
=
await
lensHub . getFollowNFT ( 1 ,
{ from :
governance });
const
followNFT
=
await
FollowNFT . at ( followNFTAddr );
const
totalSupply
=
await
followNFT . totalSupply ({ from :
user });
const
```

ownerOf

=

await

followNFT . ownerOf (1 ,

{ from :

user });

console . log (Follow NFT total supply (should be 1): { totalSupply });

console . log (

Follow NFT owner of ID 1: { ownerOf } , user address (should be the same): { user }

);

}

catch (err)

{

console . log (err);

}

cb (); } module . exports

=

main ; Replace[SecretFollowModuleAddress] in line 36 with the contract address. You can easily find it by runningtruffle networks .

If you haven't executed thecreate-profile script yet, you can uncomment the profile creation piece of this code. Otherwise, if you have created a profile, leave that portion commented, since you cannot create two profiles with the same username.

To run, simply call:

truffle exec

scripts/secret_follow.js You should see something a bit like this:

Using network 'development' .

Expected failure occurred! Error: StatusError: Transaction: 0x1c22b1e9b35d6531b22e22d807d58be55c96a81e343bf2bb3f5bd35145a1b255 exited with an error (status 0) . Reason given: Custom error (could not decode) . Please check that the transaction: - satisfies all conditions set

by Solidity require

statements.

does not trigger a Solidity revert

statement.

Follow NFT total supply (should be 1) : 1 Follow NFT owner of ID 1 : 0xA9A3b27098f4446a1019F75e1164F4ca1980727e, user address (should be the same) : 0xA9A3b27098f4446a1019F75e1164F4ca1980727e The first failure is expected because we intentionally input the wrong password!

Future extensions¶

So there you have it! We've gone over how to incorporate the Lens API into your dapp frontends and how to customize the Lens functionality by modifying their smart contracts using modules. There are a variety of ways to extend this content, such as creating a more fully fleshed dapp like Twitter or gating Lens actions through NFT ownership. Let us know how you utilized the Lens box by joining our community!

If you want to talk about this content, join ourDiscord ! If you need help coding, start a discussionhere . Lastly, don't forget to

follow us onTwitter for the latest updates on all things Truffle.