Turn Existing Contracts Into Suapps

Our goal in this tutorial is to take theCounter.sol starter contract in Forge and turn it into a fully functioning Suapp.

Setup

Ideally, this tutorial contains what you need to know to turn your existing smart contracts into Suapps. In order to demonstrate the essentials, let's start a new project and work with the standard Forge templates.

mkdir suapp &&

cd suapp forge init forge install flashbots/suave-std

Adjust Counter.sol

Delete the contents ofsrc/Counter.sol and copy the following snippet into the file:

// SPDX-License-Identifier: UNLICENSED pragma

solidity

^ 0.8.19 ;

import

"suave-std/suavelib/Suave.sol" ; import

{ Suapp }

from

"suave-std/Suapp.sol" ;

contract

Counter

is Suapp { uint256

public number ;

event

NumberSet (uint256 number) ;

modifier

confidential ( )

{ require (Suave . isConfidential ( ) , "function must be called confidentially") ; _ ; }

function

onSetNumber (uint256 newNumber)

external emitOffchainLogs { number = newNumber ; }

function

setNumber ( )

public confidential returns

(bytes

memory)

{ bytes

memory data = Suave . confidentialInputs ( ) ; uint256 newNumber = abi . decode (data ,

(uint256)) ; emit

NumberSet (newNumber) ; return abi . encodeWithSelector (this . onSetNumber . selector , newNumber) ; } } We've adjustedsetNumber and changedincrement toonSetNumber .

onSetNumber behaves like our "onchain" function from previous tutorials, and updates the number stored in our smart contract. As you'll see shortly, we don't call this function directly (though we could, but the inputs would be public).

info The "onchain" functions we've been usng throughout our docs are required to be public, which may present a challenge in certain use cases where you want to always restrict who can call them. A community contributor solved this by restricting access to methods with a modifier requiring a secret. It's calledConfidentialControl and you can see anexample usage here . setNumber is executed by a user sending a CCR and behaves like the "offchain" functions from previous tutorials.

Make sure to removetest/Counter.t.sol : we won't need it for this example. Now you can build the new contract and deploy it (make suresuave-geth is running locally so you have somewhere to deploy to):

forge build suave-geth spell deploy Counter.sol:Counter You should see an output like this:

INFO [06-26 | 18 :36:34.957] Running with local devchain settings INFO [06-26 | 18 :36:34.970] Hash of the result onchain transaction hash = 0xf92a68badb9cdd8325b5ac3de9a3892de3531ae67f0480f9bbfe14487aedd960 INFO [06-26 | 18 :36:34.970] Waiting for the transaction to be mined .. . INFO [06-26 | 18 :36:35.073] Transaction mined status = 1

blockNum

1 INFO [06-26 | 18 :36:35.073] Contract deployed address = 0xd594760B2A36467ec7F0267382564772D7b0b7 Finally, set the address your contract was deployed at as a env variable:

export

COUNTER_ADDRESS

< your_contract_address

Using the Typescript SDK

Building your own frontend for this newCounter.sol contract can take a great variety of forms. Therefore, rather than assume a framework for you, we'll demonstrate how to write generic Typescript to interact with your contract, which you can use in whichever frontend framework you prefer.

We also maintain aGolang SDK if you would prefer to use that.

Let's create anindex.ts file to demonstrate. Paste the below code into it:

import

```
{ http , decodeEventLog , encodeAbiParameters , encodeFunctionData , type
Address , type
Hex }
from
'@flashbots/suave-viem' ; import
{ getSuaveProvider , getSuaveWallet , type
TransactionRequestSuave }
from
'@flashbots/suave-viem/chains/utils' ; import
Counter
from
"./out/Counter.sol/Counter.json" ;
const
SUAVE_RPC_URL
=
'http://localhost:8545' ; const suaveProvider =
getSuaveProvider ( http ( SUAVE_RPC_URL ) ) ;
// create a wallet with the pre-funded devenet account const
PRIVATE_KEY :
Hex
= '0x91ab9a7e53c220e6210460b65a7a3bb2ca181412a8a7b43ff336b3df1737ce12' ; const wallet =
getSuaveWallet ( { transport :
http ( SUAVE_RPC_URL ) , privateKey :
PRIVATE_KEY , } ) ;
async
function
main ( )
{ if
( ! process . env . COUNTER_ADDRESS )
{ throw
new
Error ( 'COUNTER_ADDRESS env var must be set' ) ; } const counterAddress = process . env . COUNTER_ADDRESS
as
Address ;
suaveProvider . watchPendingTransactions ( { async
onTransactions ( transactions )
{ for
( const hash of transactions )
{ try
{ const receipt =
await suaveProvider . getTransactionReceipt ( { hash } ) ; console . log ( 'Transaction Receipt:' , receipt ) ; if
( receipt . status
===
'success'
&& receipt . logs . length
0 )
{ const decodedLogs =
decodeEventLog ( { abi :
Counter . abi , ... receipt . logs [ 0 ] , } ) console . log ( "decoded logs" , decodedLogs ) } }
catch
( error )
{ console . error ( 'Error fetching receipt:' , error ) ; } } } , } ) ;
const gasPrice =
await suaveProvider . getGasPrice ( ) ;
const ccr :
TransactionRequestSuave
=
{ to : counterAddress , value :
0n , gasPrice , gas :
690000n , type :
```

'0x43' , data :

encodeFunctionData ({ abi :

Counter . abi , functionName :

"setNumber" , }) , isEIP712 , confidentialInputs :

encodeAbiParameters ([{ type :

'uint256' }] ,

[13n]) , kettleAddress :

"0xB5fEAfbDD752ad52Afb7e1bD2E40432A485bBB7F" , } ;

await wallet . sendTransaction (ccr) ; }

main ( ) ; The most critical part to understand is crafting and sending Confidential Compute Requests:

1. We specify transaction type0x43
2. to indicate that this is a Confidential Compute Request.
3. We send the request to our smart contract (counterAddress
4.) to callsetNumber
5. with confidential inputs.
6. We call thesetNumber
7. function by ABI-encoding the function call in thedata
8. field, the same as you would for an ethereum transaction.
9. We specifyisEIP712
10. (a boolean which defaults totrue
11. above), to signify that the request should be signed as EIP712 signed typed data. This is particularly useful as it allows users to interact with Suave without ever changing their RPC endpoint. We recommend you always set it totrue
12. .
13. We provide our confidential data (also ABI-encoded) in theconfidentialInputs
14. field; this data is not revealed publicly, and is only known to the kettle.
15. ThekettleAddress
16. we use is specific to the local devnet. On a public testnet, this value is different. If you're looking for that address you can find ithere
17. .

info confidentialInputs is a field to store information that should be kept private during computation, and the data field is the typical calldata required to interact with a dapp.

You can see more examples of how to craft your own CCRs in theexamples directory of suave-viem . There are two ways to run this file: you can either usebun or, if you don't want to install a new tool, we can work around with Node:

- bun
- node

bun add @flashbots/suave-viem bun run index.ts Node can't run typescript off-the-bat, so we'll first need to installnode-ts :

npm i @flashbots/suave-viem npm i -D ts-node Paste this into a newtsconfig.json file:

{ "compilerOptions" :

{ "target" :

"es2020" , "module" :

"commonjs" , "strict" :

true , "esModuleInterop" :

true , "skipLibCheck" :

true , "forceConsistentCasingInFileNames" :

true , "resolveJsonModule" :

true } } Adjust your package.json to inlude"type": "commonjs" , and now you should be able to run:

npx ts-node index.ts In either case, you should see something like this printed to your terminal:

Transaction Receipt: { blockHash: "0xee533da6fad9ae95b7b3f5dd74711d011b0c984490274b3936159c5555e72e32" , blockNumber: 7n, contractAddress: null, cumulativeGasUsed: 120532n, effectiveGasPrice: 4200000000n, from: "0xbe69d72ca5f88acba033a063df5dbe43a4148de0" , gasUsed: 120532n, logs: [{ address: "0xd594760b2a36467ec7f0267382564772d7b0b73c" , topics: [

"0x9ec8254969d1974eac8c74afb0c03595b4ffe0a1d7ad8a7f82ed31b9c8542591"] , data: "0x000000000000000000000000000000000000000000000000000000000000000d" , blockNumber: 7n, transactionHash: "0x98619cc542c1aa6d427e4ce6edbb4eeef5d98c0dcf96f094fe93be28632b582b" , transactionIndex: 0 , blockHash: "0xee533da6fad9ae95b7b3f5dd74711d011b0c984490274b3936159c5555e72e32" , logIndex: 0 , removed: false, }] , logsBloom: "0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000040000000400000000000000000000000080000 , status: "success" , to: "0xd594760b2a36467ec7f0267382564772d7b0b73c" , transactionHash: "0x98619cc542c1aa6d427e4ce6edbb4eeef5d98c0dcf96f094fe93be28632b582b" , transactionIndex: 0 , type: "0x50" , } decoded logs { eventName: "NumberSet" , args: { number: 13n, } , } Congratulations! You've just built your first working Suapp!

Edit this page Previous External Calls N SUAVE Testnets