Deploy a smart contract to your Arbitrum rollup

Overview

Welcome to the guide on deploying a smart contract to your Arbitrum rollup. In this tutorial, you will learn how to deploy a smart contract using the L2 Nitro devnet and the provided public and private keys for testing purposes.

Prerequisites

- Nitro rollup devnet
- running
- Foundry
- installed on your machine
- Node.js
- Basic understanding of Ethereum
- Basic understanding of Solidity and Node.js

Setup

First, in yourHOME directory, set up a new project folder for this tutorial and init the project with npm:

bash cd HOME mkdir

counter-project && cd

counter-project && npm

init

-y cd HOME mkdir

counter-project && cd

counter-project && npm

init

-y Next, initialize a Foundry project with the following command:

bash forge

init

counter_contract forge

init

counter_contract

Create your smart contract

Take a look at theCounter.sol file in yourcounter-project/counter_contract/src directory:

solidity // SPDX-License-Identifier: UNLICENSED pragma

solidity

^0.8.13 ;

contract Counter { uint256

public number;

function

setNumber (uint256

newNumber) public { number = newNumber; }

function

increment () public { number ++ ; } } // SPDX-License-Identifier: UNLICENSED pragma

solidity

^0.8.13 ;

contract Counter { uint256

public number;

function

setNumber (uint256

newNumber) public { number = newNumber; }

function

increment () public { number ++ ; } } The contract contains a public unsigned integer variable named "number". There are two public functions in this contract. ThesetNumber function allows anyone to set a new value for the "number" variable, while theincrement function increases the value of "number" by one each time it's called.

You canlearn more about Solidity and smart contract programming .

To compile the contract, run the following forge command from theHOME/counter-project/counter_contract/ directory:

bash forge

build forge

build Your output should look similar to the following:

bash [⠒] Compiling... [⠒] Compiling 21 files with 0.8.19 [⠒] Solc 0.8.19 finished in 1.24s Compiler

run

successful [⠒] Compiling... [⠒] Compiling 21 files with 0.8.19 [⠒] Solc 0.8.19 finished in 1.24s Compiler

run

successful

Test your smart contract

Now, open thetest/Counter.t.sol file:

solidity // SPDX-License-Identifier: UNLICENSED pragma

solidity

^0.8.13 ;

import

"forge-std/Test.sol" ; import

"../src/Counter.sol" ;

contract

CounterTest

is

Test { Counter public counter;

function

setUp () public { counter =

new

Counter (); counter. setNumber (0); }

function

testIncrement () public { counter. increment (); assertEq (counter. number (), 1); }

function

testSetNumber (uint256

x) public { counter. setNumber (x); assertEq (counter. number (), x); } } // SPDX-License-Identifier: UNLICENSED pragma

solidity

^0.8.13 ;

import

"forge-std/Test.sol" ; import

"../src/Counter.sol" ;

contract

CounterTest

is

Test { Counter public counter;

function

setUp () public { counter =

new

Counter (); counter. setNumber (0); }

function

testIncrement () public { counter. increment (); assertEq (counter. number (), 1); }

function

testSetNumber (uint256

x) public { counter. setNumber (x); assertEq (counter. number (), x); } } This file performs unit testing on the contract we created in the previous section. Here's what the test is doing:

- The contract includes a public "Counter" type variable called "counter". In thesetUp
- function, it initializes a new instance of the "Counter" contract and sets the "number" variable to 0.
- There are two test functions in the contract:testIncrement
- andtestSetNumber
- .
- ThetestIncrement
- function tests the "increment" function of the "Counter" contract by calling it and then asserting that the "number" in the "Counter" contract is 1. It verifies if the increment operation correctly increases the number by one.
- ThetestSetNumber
- function is more generic. It takes an unsigned integer argument 'x' and tests the "setNumber" function of the "Counter" contract. After calling the "setNumber" function with 'x', it asserts that the "number" in the "Counter" contract is equal to 'x'. This verifies that the "setNumber" function correctly updates the "number" in the "Counter" contract.

Now, to test your code, run the following:

bash forge

test forge

test If the test is successful, your output should be similar to this:

bash [**:**] Compiling... No

files

changed,

compilation

skipped

Running

2

tests

for

test/Counter.t.sol:CounterTest [PASS] testIncrement () (gas:

28334) [PASS] testSetNumber(uint256) (runs:

256 ,

µ:

27709 ,

~:

28409) Test

result:

ok.

2

passed ; 0

failed ; finished

in

8.96 ms [⠒] Compiling... No

files

changed,

compilation

skipped

Running

2

tests

for

test/Counter.t.sol:CounterTest [PASS] testIncrement () (gas:

28334) [PASS] testSetNumber(uint256) (runs:

256 ,

µ:

27709 ,

~:

28409) Test

result:

ok.

2

passed ; 0

failed ; finished

in

8.96 ms

Deploying your smart contract

Funded accounts

Your L2 Nitro devnet will have apublic and private key funded as a faucet to use for testing:

- On both L1 and L2* Public key:0x3f1Eae7D46d88F08fc2F8ed27FCb2AB183EB2d0E
-
  - Private key:0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659

Alternatively, you canfund other addresses by using the scriptssend-l1 andsend-l2.

The L1 Geth devnet will be running athttp://localhost:8545 and the L2 Nitro devnet will be onhttp://localhost:8547 andws://localhost:8548 .

Using our Arbitrum devnet

We will use the local RPC endpoint (http://localhost:8547) and accounts above to test with.

Let's deploy the contract now. First, set a private key from anvil:

bash export L2_PRIVATE_KEY = 0xe887f7d17d07cc7b8004053fb8826f6657084e88904bb61590e498ca04704cf2 export ARB_RPC_URL = http://localhost:8547 export L2_PRIVATE_KEY = 0xe887f7d17d07cc7b8004053fb8826f6657084e88904bb61590e498ca04704cf2 export ARB_RPC_URL = http://localhost:8547 Now, deploy the contract:

bash forge

create

--rpc-url ARB_RPC_URL \ --private-key L2_PRIVATE_KEY \ src/Counter.sol:Counter forge

create

--rpc-url ARB_RPC_URL \ --private-key L2_PRIVATE_KEY \ src/Counter.sol:Counter A successful deployment will return output similar to below:

bash [⠒] Compiling... No

files

changed,

compilation

skipped Deployer:

0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 Deployed

to:

0x5FbDB2315678afecb367f032d93F642f64180aa3 Transaction

hash:

0xf1a793a793cd9fc588f5132d99008565ea361eb3535d66499575e9e1908200b2 [⠊] Compiling... No

files

changed,

compilation

skipped Deployer:

0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 Deployed

to:

0x5FbDB2315678afecb367f032d93F642f64180aa3 Transaction

hash:

0xf1a793a793cd9fc588f5132d99008565ea361eb3535d66499575e9e1908200b2 Once you've deployed the contract, you're ready to interact with it!

First, we'll set it as a variable:

bash export CONTRACT_ADDRESS = 0x5FbDB2315678afecb367f032d93F642f64180aa3 export CONTRACT_ADDRESS = 0x5FbDB2315678afecb367f032d93F642f64180aa3

Interacting with your smart contract

Foundry usescast , a CLI for performing Ethereum RPC calls.

To write to the contract, we'll use thecast send command:

bash cast

send CONTRACT_ADDRESS "setNumber(uint256)"

10

\ --rpc-url ARB_RPC_URL --private-key L2_PRIVATE_KEY cast

send CONTRACT_ADDRESS "setNumber(uint256)"

10

\ --rpc-url ARB_RPC_URL --private-key L2_PRIVATE_KEY Your output will look similar:

bash blockHash

0x131822bef6eb59656d7e1387c19b75be667e587006710365ec5cf58030786c42 blockNumber

3 contractAddress cumulativeGasUsed

43494 effectiveGasPrice

3767182372 gasUsed

43494 logs [] logsBloom

0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
root status

1 transactionHash

0x8f15d6004598f0662dd673a9898dceef77be8cc28408cecc284b28d7be32307d transactionIndex

0 type

2 blockHash

0x131822bef6eb59656d7e1387c19b75be667e587006710365ec5cf58030786c42 blockNumber

3 contractAddress cumulativeGasUsed

43494 effectiveGasPrice

3767182372 gasUsed

43494 logs [] logsBloom

0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
root status

1 transactionHash

0x8f15d6004598f0662dd673a9898dceef77be8cc28408cecc284b28d7be32307d transactionIndex

0 type

2 Now, we can make a read call to view the state of the number variable, using thecast call command:

bash cast

call CONTRACT_ADDRESS "number()"

--rpc-url ARB_RPC_URL cast

call CONTRACT_ADDRESS "number()"

--rpc-url ARB_RPC_URL The result will look similar:

bash 0x000000000000000000000000000000000000000000000000000000000000000a 0x000000000000000000000000000000000000000000000000000000000000000a Convert the result from hexadecimal to a base 10 value with:

bash echo

((0x000000000000000000000000000000000000000000000000000000000000000a)) echo

((0x000000000000000000000000000000000000000000000000000000000000000a))

Next steps

Congratulations! You've learned how to deploy a smart contract to your Arbitrum rollup devnet.

What will you build next? In our next tutorial, we will be going over how to deploy a dapp to your Arbitrum rollup.[[Edit this page on GitHub] Last updated: Previous page Nitrogen testnet Next page Deploy a dapp on your Arbitrum rollup devnet []