

Deploy a smart contract on Bubs testnet

In this tutorial, we will deploy a smart contract to the Bubs testnet.

Dependencies

- [Foundry](#)
- installed on your machine
- [Node.js](#)
- Basic understanding of Ethereum
- Basic understanding of Solidity and Node.js
- Bubs ETH from the [Bubs faucet](#)
- A Bubs RPC URL from the [Bubs testnet page](#)

Setup

First, in your HOME 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 the Counter.sol file in your counter-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. The setNumber function allows anyone to set a new value for the "number" variable, while the increment function increases the value of "number" by one each time it's called.

You can [learn more about Solidity and smart contract programming](#).

To compile the contract, run the following forge command from the HOME/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 the test/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 the setUp 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 and testSetNumber
- .
- The testIncrement 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.
- The testSetNumber 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

Using Anvil

First, we'll test out our contract on a local devnet called "anvil". To start the local server, run:

```
bash anvil anvil You'll see a local RPC endpoint (127.0.0.1:8545 ) and accounts to test with.
```

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

```
bash export PRIVATE_KEY = 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 export ANVIL_RPC_URL = http://localhost:8545 export PRIVATE_KEY = 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 export ANVIL_RPC_URL = http://localhost:8545 Now, deploy the contract:
```

```
bash forge
```

```
create
```

```
--rpc-url ANVIL_RPC_URL \ --private-key PRIVATE_KEY \ src/Counter.sol:Counter forge
```

```
create
```

```
--rpc-url ANVIL_RPC_URL \ --private-key PRIVATE_KEY \ src/Counter.sol:Counter
```

Using Bubs

First, set a private key from your funded Ethereum wallet and set theBUBS_RPC_URL variable with a[RPC of your choosing](#) :

```
bash export BUBS_PRIVATE_KEY = 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 export BUBS_RPC_URL = https://bubs.calderachain.xyz/http export BUBS_PRIVATE_KEY = 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 export BUBS_RPC_URL = https://bubs.calderachain.xyz/http Now that we're ready to deploy the smart contract onto Bubs, we will run theforge create command.
```

```
bash forge
```

```
create
```

```
--rpc-url BUBS_RPC_URL \ --private-key BUBS_PRIVATE_KEY \ src/Counter.sol:Counter forge
```

```
create
```

```
--rpc-url BUBS_RPC_URL \ --private-key BUBS_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
```

```
bash export CONTRACT_ADDRESS = 0x5FbDB2315678afecb367f032d93F642f64180aa3 export CONTRACT_ADDRESS = 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

[illegible]

Congratulations! You've learned how to deploy a smart contract to Bubs testnet.

