**Deploy a smart contract locally using Anvil**

**Deploy a smart contract locally using Anvil via scripts**

Deploying a smart contract via scripting is particularly handy because it provides a consistent and repeatable way to deploy reliably and its features enhance the testing of both the deployment processes and the code itself.

There's a strong chance that you like the command-line approach, but scripting enriches the whole deployment process, bringing in more functionality and an ease of use second to none.

Foundry eases the whole process since it is written in Solidity. This means our deployment scripts will also be written in Solidity. It is essential to distinguish Solidity as a contract language from Solidity as a scripting language. Foundry also incorporates elements that enhance our Solidity experience beyond the smart contracts realm. So, let's get started on creating a script to deploy our simple storage contract.

In Foundry we keep our scripts in the script folder.

Please create a new file called DeploySimpleStorage.s.sol.

Using .s.sol as a suffix is a naming convention for Foundry scripts, in future lessons, when we'll write Foundry tests, these will bear the suffix of .t.sol.

For more best practice info regarding Foundry scripts please click [here](https://book.getfoundry.sh/tutorials/best-practices#scripts).

Open the newly created file. Here we'll write a solidity script for deploying our SimpleStorage contract.

Type the following:

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

contract DeploySimpleStorage {

}

The first two lines are pretty self-explanatory.

We declare the new contract, named DeploySimpleStorage

For it to be considered a Foundry script and to be able to access the extended functionality Foundry is bringing to the table we need to import Script from "forge-std/Script.sol" and make DeploySimpleStorage inherit Script.

**NOTE**: forge-std also called Forge Standard Library is a collection of pre-written Solidity contracts designed to simplify and enhance scripting and testing within the Foundry development framework.

Furthermore, to be able to deploy SimpleStorage we also need to import it by typing import {SimpleStorage} from "../src/SimpleStorage.sol";

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

import {Script} from "forge-std/Script.sol";

import {SimpleStorage} from "../src/SimpleStorage.sol";

contract DeploySimpleStorage is Script {

}

Every script needs a main function, which, according to the best practice linked above is called run. Whenever you run forge script this is the function that gets called.

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

import {Script} from "forge-std/Script.sol";

import {SimpleStorage} from "../src/SimpleStorage.sol";

contract DeploySimpleStorage is Script {

function run() external returns (SimpleStorage) {

vm.startBroadcast();

SimpleStorage simpleStorage = new SimpleStorage();

vm.stopBroadcast();

return simpleStorage;

}

}

run is an external function that will return the SimpleStorage contract.

In the Run function, we are going to use a distinctive keyword: vm. Foundry has a distinctive feature known as cheat codes. The vm keyword is a cheat code in Foundry, and thereby only works in Foundry.

vm.startBroadcast indicates the starting point for the list of transactions that get to be sent to the RPC URL;

Similarly, vm.stopBroadcast indicates the ending point of the list of transactions that get to be sent to the RPC URL;

Between those two we write:

SimpleStorage simpleStorage = new SimpleStorage();

The new keyword is used to create a new smart contract in Solidity.

We end the function with return simpleStorage;.

Please select the Anvil terminal and press CTRL(CMD) + C to stop it. Now run the following command:

forge script script/DeploySimpleStorage.s.sol

This should go through without any errors, but if you hit some errors related to incompatible solidity versions in various files please ensure that both the SimpleStorage.sol and DeploySimpleStorage.s.sol use pragma solidity 0.8.19;

If you want to further extend your knowledge about scripting please go [here](https://book.getfoundry.sh/tutorials/solidity-scripting?highlight=scr#solidity-scripting)

You should get the following output:

[⠆] Compiling...

[⠔] Compiling 2 files with 0.8.19

[⠒] Solc 0.8.19 finished in 1.08s

Compiler run successful!

Script ran successfully.

Gas used: 338569

== Return ==

0: contract SimpleStorage 0x90193C961A926261B756D1E5bb255e67ff9498A1

If you wish to simulate on-chain transactions pass a RPC URL.

**The million-dollar question**: If we didn't pass an RPC URL, where did this deploy to?

If the RPC URL is not specified, Foundry automatically launches an Anvil instance, runs your script (in our case deployed the contract) and then terminates the Anvil instance.

Run the anvil command in the terminal, open up a new terminal and type the following:

forge script script/DeploySimpleStorage.s.sol --rpc-url <http://127.0.0.1:8545>

To get the following output:

No files changed, compilation skipped

EIP-3855 is not supported in one or more of the RPCs used.

Unsupported Chain IDs: 31337.

Contracts deployed with a Solidity version equal or higher than 0.8.20 might not work properly.

For more information, please see https://eips.ethereum.org/EIPS/eip-3855

Script ran successfully.

== Return ==

0: contract SimpleStorage 0x34A1D3fff3958843C43aD80F30b94c510645C316

## Setting up 1 EVM.

==========================

Chain 31337

Estimated gas price: 2 gwei

Estimated total gas used for script: 464097

Estimated amount required: 0.000928194 ETH

==========================

SIMULATION COMPLETE. To broadcast these transactions, add --broadcast and wallet configuration(s) to the previous command. See forge script --help for more.

**Another million-dollar question**: Is it deployed now?

Answer: No, the output indicates this was a simulation. But, we got a new folder out of this, the broadcast folder contains information about different script runs in case we forget details.

Hit the up arrow key and add --broadcast --private-key 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 at the end.

Or

--broadcast –interactive=<num>

Our contract is now successfully deployed! Fantastic!

Switch to the anvil terminal where you'll see:

Transaction: 0x73eb9fb4ef7b159e03c50d669c42e2ec4eeaa9358bea0a710cb07168e5192570

Contract created: 0x5fbdb2315678afecb367f032d93f642f64180aa3

Gas used: 357088

Block Number: 1

Block Hash: 0x8ea564f146e04bb36fc27f0b491223a023b5882d2fcfce3ff85e0dd152e611e4

Block Time: "Tue, 16 Apr 2024 13:39:51 +0000"

Transaction: 0x73eb9fb4ef7b159e03c50d669c42e2ec4eeaa9358bea0a710cb07168e5192570

Contract created: 0x5fbdb2315678afecb367f032d93f642f64180aa3

Gas used: 357088

Block Number: 1

Block Hash: 0x8ea564f146e04bb36fc27f0b491223a023b5882d2fcfce3ff85e0dd152e611e4

Block Time: "Tue, 16 Apr 2024 13:39:51 +0000"

Awesome!

Through this lesson we kept talking about a thing called `transaction`, but what is a `transaction`? Tune in the next lesson to find out.