Lesson 11

Foundry

Foundry is a smart contract development toolchain written in Solidity. It can:

- manage dependencies
- compile your project
- run tests
- deploy contracts
- interact with the chain via scripts and CMD

Forge is the CLI tool to initialise, build and test Foundry projects.

Foundry allows fast (milliseconds) and sophisticated Solidity tests, e.g. reading and writing directly to storage slots.

Getting Started

1. Install Foundry
    - Steps for various operation systems.
2. Initialise a project called **lets_forge**
    - `forge init lets_forge`
3. Build the project
    - `forge build`
4. Verify install has worked by running the tests
    - `forge test`

You can also use this Foundry template.

Project structure is as follows:

```
$ cd hello_foundry
$ tree . -d -L 1
.
├── lib
├── script
├── src
└── test

4 directories
```

Build command will generate the folders `out` and `cache` to store the contract artifact (ABI) and cached data, respectively.

You configure your project in the `foundry.toml` file. See ref: https://book.getfoundry.sh/reference/config/

General documentation available in the Foundry Book. **Highly recommended.**

Testing: Basics

> Smart contracts are tested using smart contracts, which is the secret to Foundry's speed since there is no additional compilation being carried out.

A smart contract e.g. MyContract.sol is tested using a file named MyContract.t.sol:

```
├── src\
│   └── MyContract.sol\
└── test\
    └── MyContract.t.sol
```

MyContract.t.sol will import the contract under test in order to access it's functions.

First Contract

Create a contract called `A.sol` and save it in `src` with the following contents:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

/// @title Encode smart contract A
/// @author Extropy.io
contract A {
    uint256 number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
```

```
    function retrieve() public view returns (uint256) {
        return number;
    }
}
```

Then create a file to test the smart contract, for example `A.t.sol` in `test` with the following contents:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

// Standard test libs
import "forge-std/Test.sol";
import "forge-std/Vm.sol";

// Contract under test
import {A} from "../src/A.sol";

contract ATest is Test {
    // Variable for contract instance
    A private a;

    function setUp() public {
        // Instantiate new contract instance
        a = new A();
    }

    function test_Log() public {
        // Various log examples
        emit log("here");
        emit log_address(address(this));
        // HEVM_ADDRESS is a special reserved address for the
VM
        emit log_address(HEVM_ADDRESS);
    }

    function test_GetValue() public {
        assertTrue(a.retrieve() == 0);
    }

    function test_SetValue() public {
        uint256 x = 123;
        a.store(x);
        assertTrue(a.retrieve() == 123);
    }

    // Define the value(s) being fuzzed as an input argument
    function test_FuzzValue(uint256 _value) public {
```

```
        // Define the boundaries for the fuzzing, in this case
0 and 99999
        _value = bound(_value, 0, 99999);
        // Call contract function with value
        a.store(_value);
        // Perform validation
        assertTrue(a.retrieve() == _value);
    }
}
```

Finally run the test with `forge test -vv` to see the results and all logs.

Run tests

```
forge test
```

To print event logs:

```
forge test -vv
```

To print trace of any failed tests:

```
forge test -vvv
```

To print trace of all tests:

```
forge test -vvvv
```

To run a specific test:

```
forge test --match-test test_myTest
```

Dependencies

Forge uses git submodules so it works with any GitHub repo that contains smart contracts.

Adding Dependencies

To install e.g. OpenZeppelin contracts from the repo
one would run
`forge install OpenZeppelin/openzeppelin-contracts`.
The repo will have been cloned into the `lib` folder.

Dependencies can be updated by running `forge update`.

Removing Dependencies

Dependencies can be removed by running
`forge remove openzeppelin-contracts`,
which is equivalent to
`forge remove OpenZeppelin/openzeppelin-contracts`.

Integrating with Existing Hardhat project

Foundry can work with Hardhat-style projects where dependencies are npm packages (stored in `node_modules`) and contracts are stored in `contracts` as opposed to source.

1. Copy lib/forge-std from a newly-created empty Foundry project to this Hardhat project directory.
2. Copy foundry.toml configuration to this Hardhat project directory and change src, out, test, cache_path in it:

```
[default]
src = 'contracts'
out = 'out'
libs = ['node_modules', 'lib']
test = 'test/foundry'
cache_path  = 'forge-cache'
```

3. Create a remappings.txt to make Foundry project work well with VS Code Solidity extension:

```
ds-test/=lib/forge-std/lib/ds-test/src/
forge-std/=lib/forge-std/src/
```

4. Make a sub-directory test/foundry and write Foundry tests in it.

Foundry test works in this existing Hardhat project. As the Hardhat project is not touched and it can work as before.

See folder `hardhat-foundry`. Both test suites function:

`npx hardhat test`

`forge test -vvv`

Deploying Contracts

`forge create` allows you to deploy contracts to a blockchain.

To deploy the previous contract to e.g. Ganache using an account with private key `4e0...9b` you would type:

`forge create A --legacy --contracts src/A.sol --private-key 4e0...9b --rpc-url http://127.0.0.1:8545`

Result:

```
Deployer: 0x536f8222c676b6566ff34e539022de6c1c22cc06
Deployed to: 0x79bb7a73d02b6d7e2e98848d26ad911720421df0
Transaction hash:
0xc5bb34ee82dc2f57bd7f7862eec440576a7cc7cfe4533392192704fd44653b68
```

The `--legacy` flag is used since Ganache doesn't support EIP-1559 transactions. Hardhat (`npx hardhat node`) circumvents this issue.

Solidity Scripting

> Solidity scripting is a way to declaratively deploy contracts using Solidity, instead of using the more limiting and less user friendly forge create.

Similar to Hardhat scripting but faster with dry-run capabilities (because it runs in Foundry EVM).

Scripts go in the `/scripts` folder and file extension is `.s.sol`.

Running a script. You should add the variables to the .env beforehand & run from root:

```
# To load the variables in the .env file
source .env

# To deploy and verify our contract
forge script script/NFT.s.sol:MyScript --rpc-url
```

```
$GOERLI_RPC_URL --broadcast --verify -vvvv
```

Example of a deployment script:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.18;

import "forge-std/Script.sol"; // Import script lib
import "../src/ContractA.sol";
import "../src/ContractB.sol";
import "../src/ContractC.sol";

contract Deployment is Script {
    function run() public {

        //Load key/unlock wallet. Use with caution if in
production
        uint256 deployerPrivateKey = vm.envUint("DEPLOY_KEY");
        vm.startBroadcast(deployerPrivateKey);

        //Deploy contracts
        ContractA contractA = new ContractA();
        ContractB contractB = new ContractB();
        ContractC contractC = new ContractC();

        //Do some complex deployment logic
        contractA.callSomeFunction();
        uint256 result =
contractB.anotherFunction(address(contractA));
        contractC.someOtherFunction(result);

        vm.stopBroadcast();
    }
}
```

Cast

> Perform Ethereum RPC calls from the comfort of your command line.

For example:

- `cast chain-id` Get the Ethereum chain ID.
- `cast publish` Publish a raw - transaction to the network.
- `cast receipt` Get the transaction receipt for a transaction.
- `cast send` Sign and publish a transaction.
- `cast call` Perform a call on an account without publishing a transaction.
- `cast block-number` Get the latest block number.
- `cast abi-encode` ABI encode the given function arguments, excluding the selector.

How to use Cast

```
cast [options] command [args] cast [options] --version cast
[options] --help
```

eg:

```
cast call 0x79bb7a73d02b6d7e2e98848d26ad911720421df0
"retrieve()" --rpc-url http://127.0.0.1:8545
```

There are **a lot** of cast commands. Take a look at the reference:
https://book.getfoundry.sh/reference/cast/cast

Debugging in Foundry

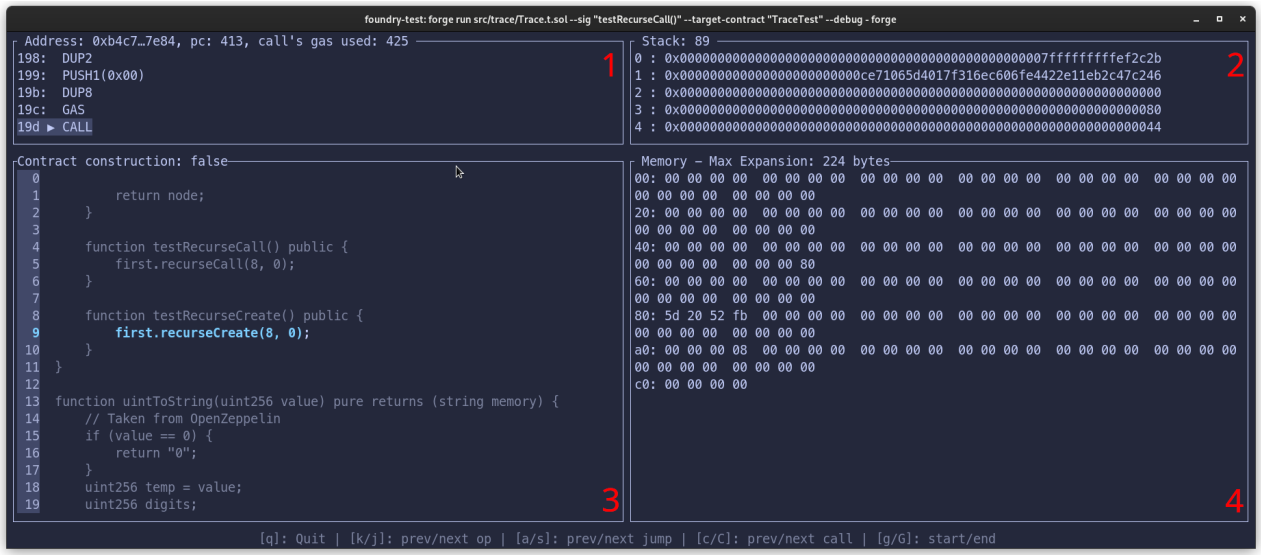
There are 2 ways to debug:

- With `forge test` passing a function from tests `forge test --debug $FUNC`

Example `forge test --debug "testSomething()"`

- With `forge debug` passing a file `forge debug --debug $FILE --sig $FUNC`

Example: `forge debug --debug src/SomeContract.sol --sig "myFunc(uint256,string)" 123 "hello"`



When the debugger is run, you are presented with a terminal divided into four quadrants:

- 1: The opcodes in the debugging session
- 2: The current stack, as well as the size of the stack
- 3: The source view
- 4: The current memory of the EVM

> If you have multiple contracts with the same function name, you need to limit the matching functions down to only one case using --match-path and --match-contract.

> If the matching test is a fuzz test, the debugger will open the first failing fuzz scenario, or the last successful one, whichever comes first.

See guide for more info book

Testing: Advanced

Fuzzing

Sometimes it's useful to test a function with many input variables at random in order to test edge-cases. This is called `fuzz testing`, or simply `fuzzing`.

An example of fuzzing using Foundry has been provided in the smart contract:

```
// Define the value(s) being fuzzed as an input argument
function test_FuzzValue(uint256 _value) public {
    // Define the boundaries for the fuzzing, in this case 0
and 99999
    _value = bound(_value, 0, 99999);
    // Call contract function with value
    a.store(_value);
    // Perform validation
    assertTrue(a.retrieve() == _value);
}
```

Interpreting results

Results may appear similar to this:

```
Running 4 tests for test/A.t.sol:ATest
[PASS] test_FuzzValue(uint256) (runs: 256, μ: 31708, ~: 32330)
[PASS] test_GetValue() (gas: 7546)
[PASS] test_Log() (gas: 3930)
Logs:
here
0xb4c79dab8f259c7aee6e5b2aa729821864227e84
0x7109709ecfa91a80626ff3989d68f67f5b1dd12d
```

- "runs" refers to the amount of scenarios the fuzzer tested. By default, the fuzzer will generate 256 scenarios, however, this can be configured using the FOUNDRY_FUZZ_RUNS environment variable.
- "μ" (Greek letter mu) is the mean gas used across all fuzz runs.
- "~" (tilde) is the median gas used across all fuzz runs.

Forking

Forge supports testing in a forked environment with two different approaches:

- Forking Mode — use a single fork for all your tests via the forge test `--fork-url` flag.

```
forge test --fork-url <your_rpc_url>
```

- Forking Cheatcodes in tests — create, select, and manage multiple forks directly in Solidity test code via forking cheatcodes

```
contract ForkTest is Test {
    // the identifiers of the forks
    uint256 mainnetFork;
    uint256 optimismFork;

    //Access variables from .env file via
vm.envString("varname")
    //Replace ALCHEMY_KEY by your alchemy key or Etherscan key,
change RPC url if need
    //inside your .env file e.g:
    //MAINNET_RPC_URL = 'https://eth-
mainnet.g.alchemy.com//v2/ALCHEMY_KEY'
    //string MAINNET_RPC_URL = vm.envString("MAINNET_RPC_URL");
    //string OPTIMISM_RPC_URL =
vm.envString("OPTIMISM_RPC_URL");

    // create two _different_ forks during setup
    function setUp() public {
        mainnetFork = vm.createFork(MAINNET_RPC_URL);
        optimismFork = vm.createFork(OPTIMISM_RPC_URL);
    }

    // demonstrate fork ids are unique
    function testForkIdDiffer() public {
        assert(mainnetFork != optimismFork);
    }

    // select a specific fork
    function testCanSelectFork() public {
        // select the fork
```

```
        vm.selectFork(mainnetFork);
        assertEq(vm.activeFork(), mainnetFork);

        // from here on data is fetched from the `mainnetFork`
if the EVM requests it and written to the storage of
`mainnetFork`
    }

    // manage multiple forks in the same test
    function testCanSwitchForks() public {
        vm.selectFork(mainnetFork);
        assertEq(vm.activeFork(), mainnetFork);

        vm.selectFork(optimismFork);
        assertEq(vm.activeFork(), optimismFork);
    }

    // forks can be created at all times
    function testCanCreateAndSelectForkInOneStep() public {
        // creates a new fork and also selects it
        uint256 anotherFork =
vm.createSelectFork(MAINNET_RPC_URL);
        assertEq(vm.activeFork(), anotherFork);
    }

    // set `block.timestamp` of a fork
    function testCanSetForkBlockTimestamp() public {
        vm.selectFork(mainnetFork);
        vm.rollFork(1_337_000);

        assertEq(block.number, 1_337_000);
    }
}
```

Cheatcodes

Cheatcodes allow you to change the block number, your identity, and more. They are invoked by calling specific functions on a specially designated address: `0x7109709ECfa91a80626fF3989D68f67F5b1DD12D`.

You can access cheatcodes easily via the 'vm' instance available in Forge Standard Library's Test contract. Forge Standard Library is explained in greater detail in the following section.

Below are some subsections for the different Forge cheatcodes.

- Environment: Cheatcodes that alter the state of the EVM.
- Assertions: Cheatcodes that are powerful assertions
- Fuzzer: Cheatcodes that configure the fuzzer
- External: Cheatcodes that interact with external state (files, commands, ...)
- Utilities: Smaller utility cheatcodes
- Forking: Forking mode cheatcodes
- Snapshots: Snapshot cheatcodes
- File: Cheatcodes for working with files

See cheatcodes reference in docs:
https://book.getfoundry.sh/cheatcodes/

```
// Set block.timestamp
function warp(uint256) external;

// Set block.number
function roll(uint256) external;

// Set block.basefee
function fee(uint256) external;

// Set block.difficulty
function difficulty(uint256) external;

// Set block.chainid
function chainId(uint256) external;
```

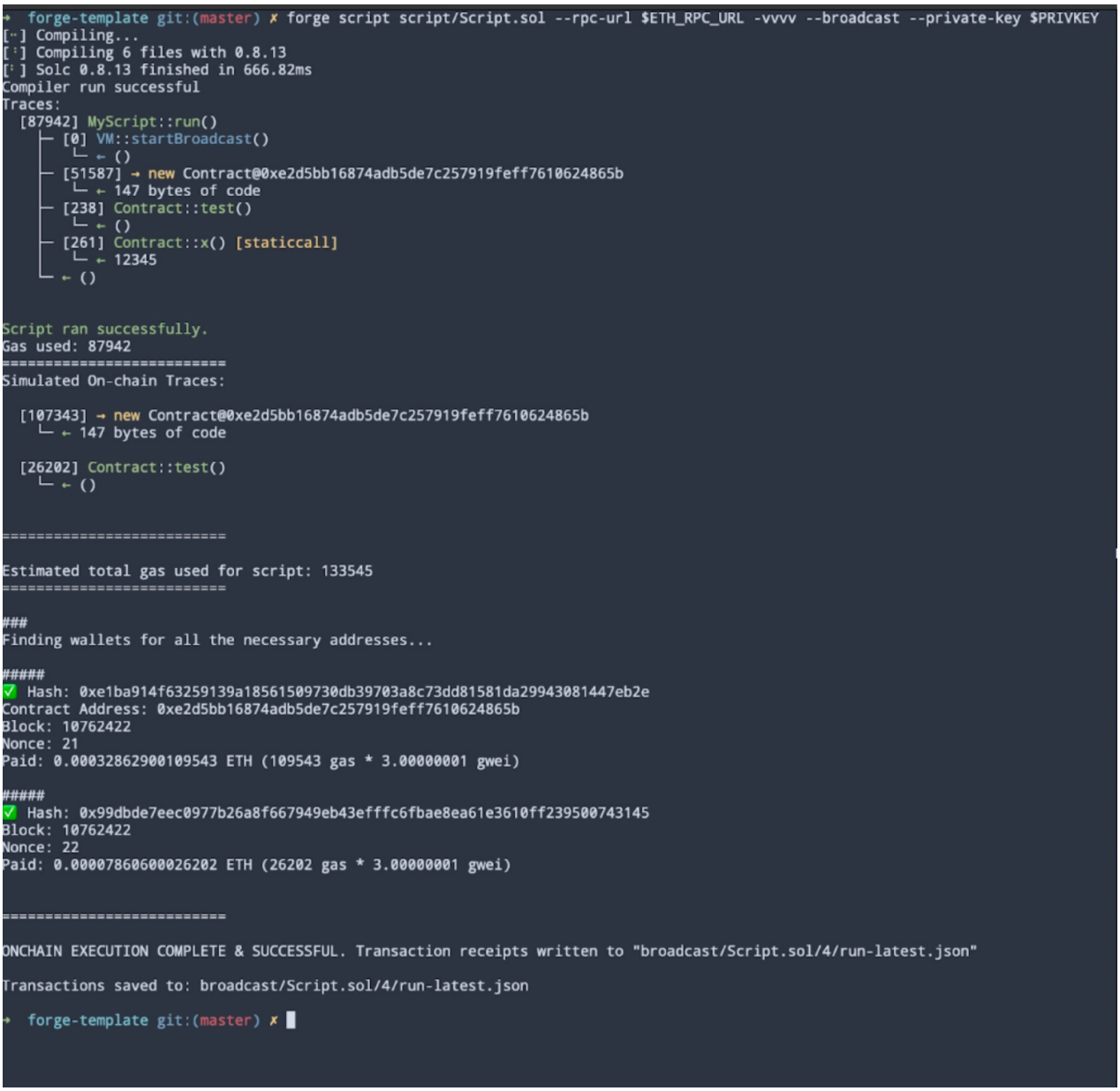
```
// Loads a storage slot from an address
function load(address account, bytes32 slot) external returns
(bytes32);

// Stores a value to an address' storage slot
function store(address account, bytes32 slot, bytes32 value)
external;

// Signs data
function sign(uint256 privateKey, bytes32 digest)
    external
    returns (uint8 v, bytes32 r, bytes32 s);

// Computes address for a given private key
function addr(uint256 privateKey) external returns (address);
```

Broadcast will emulate your deployment on chain and return a comprehensive trace of the execution.


forge-template git:(master) ✗ forge script script/Script.sol --rpc-url $ETH_RPC_URL -vvvv --broadcast --private-key $PRIVKEY
[⠒] Compiling...
[⠢] Compiling 6 files with 0.8.13
[⠆] Solc 0.8.13 finished in 666.82ms
Compiler run successful
Traces:
[87942] MyScript::run()
├─ [0] VM::startBroadcast()
│ └─ ← ()
├─ [51587] → new Contract@0xe2d5bb16874adb5de7c257919feff7610624865b
│ └─ ← 147 bytes of code
├─ [238] Contract::test()
│ └─ ← ()
├─ [261] Contract::x() [staticcall]
│ └─ ← 12345
└─ ← ()
Script ran successfully.
Gas used: 87942
==========
Simulated On-chain Traces:
[107343] → new Contract@0xe2d5bb16874adb5de7c257919feff7610624865b
└─ ← 147 bytes of code
[26202] Contract::test()
└─ ← ()
==========
Estimated total gas used for script: 133545
==========
###
Finding wallets for all the necessary addresses...
#####
✅ Hash: 0xe1ba914f63259139a18561509730db39703a8c73dd81581da29943081447eb2e
Contract Address: 0xe2d5bb16874adb5de7c257919feff7610624865b
Block: 10762422
Nonce: 21
Paid: 0.00032862900109543 ETH (109543 gas * 3.00000001 gwei)
#####
✅ Hash: 0x99dbde7eec0977b26a8f667949eb43efffc6fbae8ea61e3610ff239500743145
Block: 10762422
Nonce: 22
Paid: 0.00007860600026202 ETH (26202 gas * 3.00000001 gwei)
==========
ONCHAIN EXECUTION COMPLETE & SUCCESSFUL. Transaction receipts written to "broadcast/Script.sol/4/run-latest.json"
Transactions saved to: broadcast/Script.sol/4/run-latest.json
forge-template git:(master) ✗

Vanity Addresses

Foundry has a vanity address tool.

`cast wallet vanity [options]`

Options:

`--starts-with hex`
Prefix for the vanity address.

`--ends-with hex`
Suffix for the vanity address.

`--nonce nonce`
Generate a vanity contract address created by the generated keypair with the specified nonce.

Usage:

`cast wallet vanity --starts-with beef`

```
Address: 0xBeEFbE97aa3D0F779711B0643DBA1542a927F3A1

Private Key: 0x.....
```

Useful Foundry references:

- https://0xkowloon.substack.com/p/learn-enough-foundry-to-be-dangerous?s=r
- https://chainstack.com/foundry-a-fast-solidity-contract-development-toolkit/
- https://book.getfoundry.sh/
- https://github.com/dabit3/foundry-cheatsheet
- https://mirror.xyz/horsefacts.eth/Jex2YVaO65dda6zEyfM_-DXlXhOWCAoSpOx5PLocYgw