Building dApps on Your Own L1 with Foundry

Louis - 17/10/24

Writing Smart Contracts

- Follow the best code standards, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/GUIDELINES.md
- Use trusted libraries to make your contracts more easily
  - OpenZeppelin
  - Solady
  - Solmate
- To install them, forge install <github_account>/<repo_name>

Writing Smart Contracts

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

UnitTest stub | dependencies | uml | funcSigs | draw.io
contract Workshop is ERC20 {
    uint256 public constant MAX_SUPPLY = 9_000e18;
    uint256 public constant PRICE = 0.001e18;
    uint256 public constant PRECISION = 1e18;

    event Mint(address indexed sender, address indexed to, uint256 amount);
    event Burn(address indexed sender, address indexed from, uint256 amount);

    ftrace
    constructor() ERC20("Workshop", "WORK") {}
```

Writing Smart Contracts

```
function mint(address to, uint256 amount) public payable {
    require(amount <= MAX_SUPPLY, "Max supply exceeded");
    require(amount * PRICE / PRECISION == msg.value, "Invalid amount");

    _mint(to, amount);

    emit Mint(msg.sender, to, amount);
}

ftrace | funcSig
function burn(uint256 amount) public {
    _burn(msg.sender, amount);

    payable(msg.sender).transfer(amount * PRICE / PRECISION);

    emit Burn(msg.sender, msg.sender, amount);
}
```

Fixing your Smart Contracts

```
function mint(address to⬆, uint256 amount⬆) public payable {
    require(totalSupply() + amount⬆ <= MAX_SUPPLY, "Max supply exceeded");
    require(_divRoundUp(amount⬆ * PRICE, PRECISION) == msg.value, "Invalid amount");

    _mint(to⬆, amount⬆);

    emit Mint(msg.sender, to⬆, amount⬆);
}

ftrace | funcSig
function _divRoundUp(uint256 a⬆, uint256 b⬆) internal pure returns (uint256) {
    return a⬆ / b⬆ + (a⬆ % b⬆ == 0 ? 0 : 1);
}
```

Writing Smart Contracts

- Update your foundry.toml and add the remappings, rpc and block explorers parameters:

```
[profile.default]
src = "src"
out = "out"
libs = ["lib"]

remappings = [
    "@openzeppelin/contracts/=lib/openzeppelin-contracts/contracts/",
]

[rpc_endpoints]
avalanche = "https://api.avax.network/ext/bc/C/rpc"
fuji = "https://api.avax-test.network/ext/bc/C/rpc"

[etherscan]
avalanche = { key = "${SNOWSCAN_KEY}", url = "https://api.snowscan.xyz/api" }
fuji = { key = "${SNOWSCAN_KEY}", url = "https://api-testnet.snowscan.xyz/api" }

# See more config options https://github.com/foundry-rs/foundry/blob/master/crates/config/README.md#all-
```

Testing your Smart Contracts

- Smart Contracts should be tested, coverage should be close to 100%
  - The coverage can be checked using: forge coverage
- Tests are written in solidity, and their function name should start by « test »
- Unit tests are not enough, fuzzing should be added. And invariant testing when possible
- Tests can be ran using: forge test -vvv

Testing your Smart Contracts

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

import "forge-std/Test.sol";

import "src/WorkshopBad.sol";

UnitTest stub | dependencies | uml | funcSigs | draw.io
contract WorkshopTest is Test {
    Workshop workshop;

    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    event Mint(address indexed sender, address indexed to, uint256 amount);
    event Burn(address indexed sender, address indexed from, uint256 amount);

    ftrace | funcSig
    function setUp() public {
        workshop = new Workshop();

        payable(alice).transfer(100e18);
        payable(bob).transfer(100e18);
    }
```

Testing your Smart Contracts

```
function test_Mint() public {
    uint256 price = workshop.PRICE();

    uint256 amountAlice = 10e18;
    uint256 valueAlice = amountAlice * price / 1e18;

    vm.expectEmit(true, true, true, true);
    emit Mint(alice, alice, amountAlice);

    vm.prank(alice);
    workshop.mint{value: valueAlice}(alice, amountAlice);

    assertEq(workshop.balanceOf(alice), amountAlice, "test_Mint::1");
    assertEq(address(workshop).balance, valueAlice, "test_Mint::2");
    assertEq(address(alice).balance, 100e18 - valueAlice, "test_Mint::3");

    uint256 amountBob = 20e18;
    uint256 valueBob = amountBob * price / 1e18;

    vm.expectEmit(true, true, true, true);
    emit Mint(bob, bob, amountBob);

    vm.prank(bob);
    workshop.mint{value: valueBob}(bob, amountBob);

    assertEq(workshop.balanceOf(bob), amountBob, "test_Mint::4");
    assertEq(address(workshop).balance, valueAlice + valueBob, "test_Mint::5");
    assertEq(address(bob).balance, 100e18 - valueBob, "test_Mint::6");
}
```

Testing your Smart Contracts

```
function test_revert_Mint() public {
    uint256 maxSupply = workshop.MAX_SUPPLY();

    vm.expectRevert("Max supply exceeded");
    workshop.mint(alice, maxSupply + 1);

    uint256 price = workshop.PRICE();

    uint256 amountAlice = 10e18;
    uint256 valueAlice = amountAlice * price / 1e18;

    vm.expectRevert("Invalid amount");
    workshop.mint{value: valueAlice - 1}(alice, amountAlice);

    vm.expectRevert("Invalid amount");
    workshop.mint{value: valueAlice + 1}(alice, amountAlice);
}
```

Testing your Smart Contracts

```
function test_Burn() public {
    uint256 price = workshop.PRICE();

    uint256 amountAlice = 10e18;
    uint256 valueAlice = amountAlice * price / 1e18;
    uint256 amountBob = 20e18;
    uint256 valueBob = amountBob * price / 1e18;

    vm.prank(alice);
    workshop.mint{value: valueAlice}(alice, amountAlice);

    vm.prank(bob);
    workshop.mint{value: valueBob}(bob, amountBob);

    vm.expectEmit(true, true, true, true);
    emit Burn(alice, alice, amountAlice / 2);
```

Testing your Smart Contracts

```
function test_Burn() public {
    vm.expectEmit(true, true, true, true);
    emit Burn(alice, alice, amountAlice / 2);

    vm.prank(alice);
    workshop.burn(amountAlice / 2);

    assertEq(workshop.balanceOf(alice), amountAlice / 2, "test_Burn::1");
    assertEq(address(workshop).balance, valueAlice / 2 + valueBob, "test_Burn::2");
    assertEq(address(alice).balance, 100e18 - valueAlice / 2, "test_Burn::3");

    vm.expectEmit(true, true, true, true);
    emit Burn(bob, bob, amountBob);

    vm.prank(bob);
    workshop.burn(amountBob);

    assertEq(workshop.balanceOf(bob), 0, "test_Burn::4");
    assertEq(address(workshop).balance, valueAlice / 2, "test_Burn::5");
    assertEq(address(bob).balance, 100e18, "test_Burn::6");
}
```

Testing your Smart Contracts

```
function test_revert_Burn() public {
    uint256 price = workshop.PRICE();

    uint256 amountAlice = 10e18;
    uint256 valueAlice = amountAlice * price / 1e18;

    vm.prank(alice);
    workshop.mint{value: valueAlice}(alice, amountAlice);

    vm.prank(alice);
    vm.expectRevert(
        abi.encodeWithSelector(IERC20Errors.ERC20InsufficientBalance.selector, alice, amountAlice, amountAlice + 1)
    );
    workshop.burn(amountAlice + 1);
}
```

Testing your Smart Contracts

```
function test_fuzz_Mint(uint256 mint0↑, uint256 mint1↑) public {
    uint256 maxAmount = workshop.MAX_SUPPLY();
    uint256 price = workshop.PRICE();

    mint0↑ = bound(mint0↑, 0, maxAmount);
    maxAmount -= mint0↑;

    uint256 value0 = mint0↑ * price / 1e18;

    vm.prank(alice);
    workshop.mint{value: value0}(alice, mint0↑);

    mint1↑ = bound(mint1↑, 0, maxAmount);
    maxAmount -= mint1↑;

    uint256 value1 = mint1↑ * price / 1e18;

    vm.prank(bob);
    workshop.mint{value: value1}(bob, mint1↑);

    uint256 minted = workshop.totalSupply();
    uint256 expectedValue = minted * price / 1e18;

    assertGe(address(workshop).balance, expectedValue, "test_fuzz_Mint::1");
    assertEq(workshop.balanceOf(alice), mint0↑, "test_fuzz_Mint::2");
    assertEq(workshop.balanceOf(bob), mint1↑, "test_fuzz_Mint::3");
}
```

Testing your Smart Contracts

```
function test_fuzz_revert_Mint(uint256 amount⬆) public {
    uint256 maxAmount = workshop.MAX_SUPPLY();
    uint256 price = workshop.PRICE();

    amount⬆ = bound(amount⬆, 0, maxAmount);
    maxAmount -= amount⬆;

    uint256 value0 = amount⬆ * price / 1e18;

    vm.prank(alice);
    workshop.mint{value: value0}(alice, amount⬆);

    vm.expectRevert("Max supply exceeded");
    vm.prank(bob);
    workshop.mint{value: value0}(bob, maxAmount + 1);
}
```

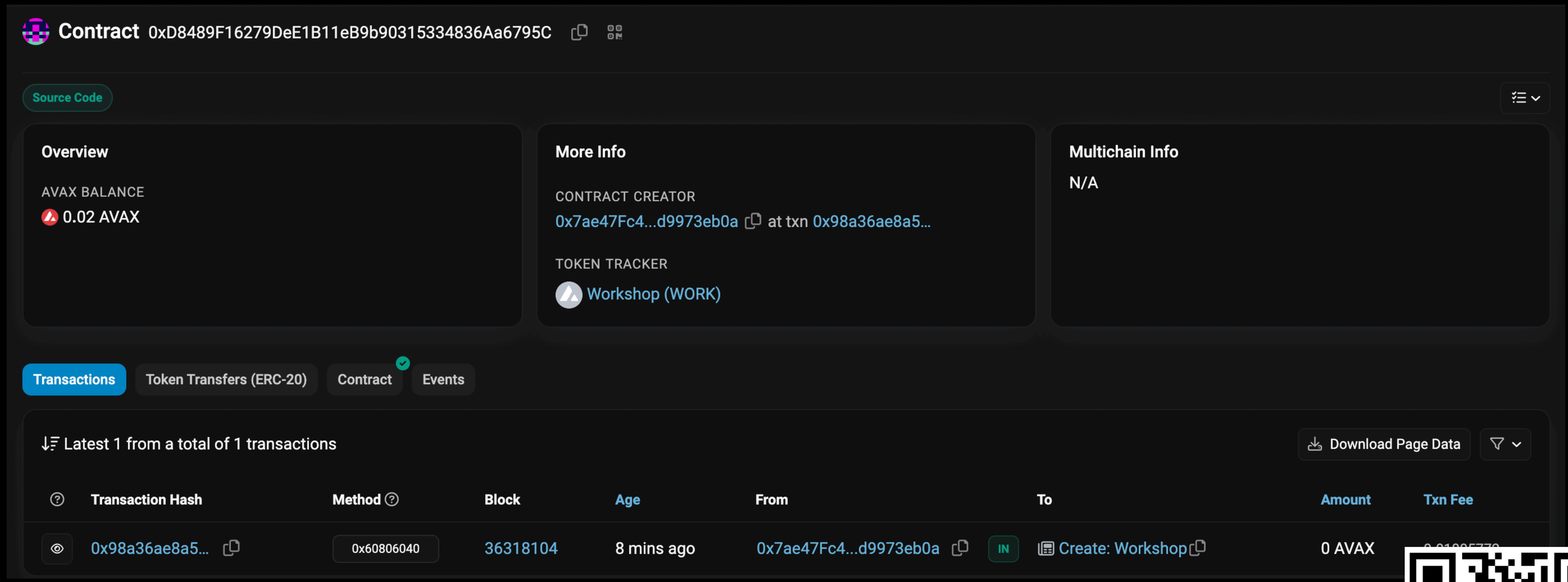
Deploy your Smart Contracts

- Smart Contracts should be deployed using scripts
- Scripts are written in solidity and should always be name « run » (only one per file)
- They should be verified on the main explorer
- Your foundry.toml and .env files should be updated with the right values
- To run a script use: forge script <path>/<script_name> --broadcast --verify

Deploy your Smart Contracts

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

import "forge-std/Script.sol";

import "src/WorkshopFixed.sol";

UnitTest stub | dependencies | uml | funcSigs | draw.io
contract WorkshopScript is Script {
    ftrace | funcSig
    function setUp() public {
        vm.createSelectFork(vm.rpcUrl("fuji"));
    }

    ftrace | funcSig
    function run() public returns (Workshop workshop) {
        uint256 pk = vm.envUint("PRIVATE_KEY");
        vm.broadcast(pk);
        workshop = new Workshop();
    }
}
```

Deploy your Smart Contracts



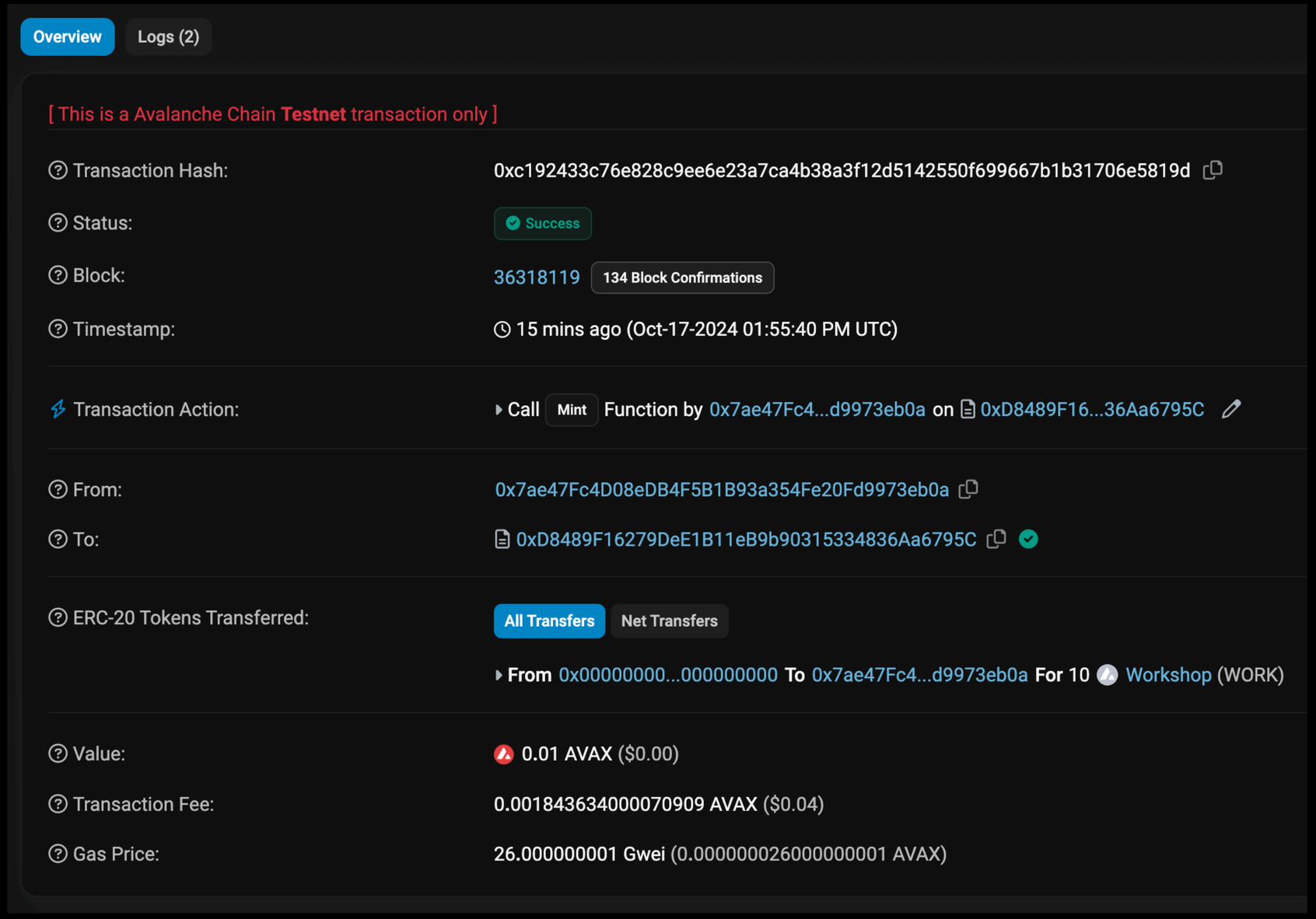
Interact with your Smart Contracts

- You should interact with smart contracts using scripts as it allows you to simulate and test the interactions
- Similarly, your interaction scripts should be written in solidity and be named « run »
- To run it, use: forge script <path>/<script_name> --broadcast

Interact with your Smart Contracts

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

import "forge-std/Script.sol";

import "src/WorkshopFixed.sol";

UnitTest stub | dependencies | uml | funcSigs | draw.io
contract WorkshopMintScript is Script {
    Workshop public constant workshop = Workshop(0xD8489F16279DeE1B11eB9b90315334836Aa6795C);

    ftrace | funcSig
    function setUp() public {
        vm.createSelectFork(vm.rpcUrl("fuji"));
    }

    ftrace | funcSig
    function run() public {
        uint256 pk = vm.envUint("PRIVATE_KEY");
        address deployer = vm.addr(pk);

        uint256 price = workshop.PRICE();

        uint256 amount = 10e18;
        uint256 value = amount * price / 1e18;

        vm.broadcast(pk);
        workshop.mint{value: value}(deployer, amount);
    }
}
```

Interact with your Smart Contracts



Rerun failed transactions

- To debug a reverted transaction, the easiest way is to use cast to rerun the transaction and show the trace. Tenderly can also be useful.
- To rerun a failed transaction, use: cast run <tx_hash> --rpc-url <rpc-url>

Rerun failed transactions

[This is a Avalanche Chain **Testnet** transaction only]

Transaction Hash:	0x4f574df8868d7cdf24242d4b4dc977dc77def4f3bca4bc41c3fa97b5ae1417e8
Status:	Fail
Block:	36318164 1 Block Confirmation
Timestamp:	12 secs ago (Oct-17-2024 02:00:44 PM UTC)
Transaction Action:	Call Mint Function by 0x7ae47Fc4...d9973eb0a on 0xD8489F16...36Aa6795C
From:	0x7ae47Fc4D08eDB4F5B1B93a354Fe20Fd9973eb0a
To:	0xD8489F16279DeE1B11eB9b90315334836Aa6795C Warning! Error encountered during contract execution [**execution reverted**]
Value:	1 AVAX ($0.00) - [CANCELLED]
Transaction Fee:	0.000616900000024676 AVAX ($0.01)
Gas Price:	25.000000001 Gwei (0.000000025000000001 AVAX)

Rerun failed transactions

```
Traces:
  [3104] 0xD8489F16279DeE1B11eB9b90315334836Aa6795C::mint{value:
    └─ ← [Revert] revert: Invalid amount
```

```
function mint(address to⬆, uint256 amount⬆) public payable {
    require(totalSupply() + amount⬆ <= MAX_SUPPLY, "Max supply exceeded");
    require(_divRoundUp(amount⬆ * PRICE, PRECISION) == msg.value, "Invalid amount");

    _mint(to⬆, amount⬆);

    emit Mint(msg.sender, to⬆, amount⬆);
}
```

Selectors of a Smart Contracts

- forge selector list

```
Workshop
+----------+-----------------------------------------------+------------+
| Type     | Signature                                     | Selector   |
+=============================================================================+
| Function | MAX_SUPPLY()                                  | 0x32cb6b0c |
|----------+-----------------------------------------------+------------|
| Function | PRECISION()                                   | 0xaaf5eb68 |
|----------+-----------------------------------------------+------------|
| Function | PRICE()                                       | 0x8d859f3e |
|----------+-----------------------------------------------+------------|
| Function | allowance(address,address)                    | 0xdd62ed3e |
|----------+-----------------------------------------------+------------|
| Function | approve(address,uint256)                      | 0x095ea7b3 |
|----------+-----------------------------------------------+------------|
| Function | balanceOf(address)                            | 0x70a08231 |
```

- forge selectors upload --all