Contract to Contract Interactions

Contract to Contract Interaction

Copy page

Interact with other smart contracts

In this article, you'll learn how to interact with other smart contracts using interfaces and the `.call()` function, which allows you to interact with other smart contracts without using an interface.

⚠ This tutorial has been moved as part of a reorganization! It assumes you are using Hardhat. Everything in this lesson will work with minor adjustments if you are working in Foundry or Remix.

Objectives

By the end of this lesson you should be able to:

Use interfaces to allow a smart contract to call functions in another smart contract

Use the `call()` function to interact with another contract without using an interface

Overview

Interacting with external smart contracts is a very common task in the life of a smart contract developer. This includes interacting with contracts that are already deployed to a particular network.

Usually the creators of certain smart contracts document their functionality and expose their functions by providing interfaces that can be used to integrate those particular contracts into your own.

For instance, **Uniswap** provides documentation on how to interact with their smart contracts and also some packages to easily integrate their protocol.

In this example, you interact with the **Uniswap protocol** to create a custom pool for a custom pair of tokens.

Since the Uniswap protocol is already deployed, you will use **Hardhat forking** to test your contract.

You will also use the following two approaches in the example:

- Using interfaces
- Using the `.call()` function

Interacting with deployed contracts using interfaces

You must first install the **Uniswap V3 core package** by running:

```
npm install @uniswap/v3-core
```

This package provides access to the Uniswap interfaces of the Core protocol.

Then, write a custom contract called `PoolCreator` with the following code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

import "@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol";

contract PoolCreator {
    IUniswapV3Factory public uniswapFactory;

    constructor(address _factoryAddress) {
        uniswapFactory = IUniswapV3Factory(_factoryAddress);
    }

    function createPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) external returns (address poolAddress) {
        // Check if a pool with the given tokens and fee already exists
        poolAddress = uniswapFactory.getPool(tokenA, tokenB, fee);
        if (poolAddress == address(0)) {
            // If the pool doesn't exist, create a new one
            poolAddress = uniswapFactory.createPool(tokenA, tokenB, fee);
        }

        return poolAddress;
    }
}
```

Notice the following:

- You are importing a `IUniswapV3Factory` interface. The interface contains function declarations that include `getPool` and `createPool`:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity >=0.5.0;

/// @title The interface for the Uniswap V3 Factory
/// @notice The Uniswap V3 Factory facilitates creation of Uniswap V3 pools
interface IUniswapV3Factory {
    // ...
    // ...other function declarations

    /// @notice Returns the pool address for a given pair of tokens and a f
    /// @dev tokenA and tokenB may be passed in either token0/token1 or tok
    /// @param tokenA The contract address of either token0 or token1
    /// @param tokenB The contract address of the other token
    /// @param fee The fee collected upon every swap in the pool, denominat
    /// @return pool The pool address
    function getPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) external view returns (address pool);

    /// @notice Creates a pool for the given two tokens and fee
    /// @param tokenA One of the two tokens in the desired pool
    /// @param tokenB The other of the two tokens in the desired pool
    /// @param fee The desired fee for the pool
    /// @dev tokenA and tokenB may be passed in either order: token0/token1
    /// from the fee. The call will revert if the pool already exists, the
    /// are invalid.
    /// @return pool The address of the newly created pool
    function createPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) external returns (address pool);
```

> The constructor receives the address of the pool factory and creates an instance of `IUniswapV3Factory` .
>
> The `createPool` function includes a validation to ensure the pool doesn't exist.
>
> The `createPool` function creates a new pool.

Then, create a test file called `PoolCreator.test.ts` with the following content:

```
import { ethers } from 'hardhat';
import { HardhatEthersSigner } from '@nomicfoundation/hardhat-ethers/signer

import { Token, Token__factory, PoolCreator, PoolCreator__factory } from '.

describe('PoolCreator tests', function () {
  const UNISWAP_FACTORY_ADDRESS = '0x1F98431c8aD98523631AE4a59f267346ea31F9
  let tokenA: Token;
  let tokenB: Token;
  let poolCreator: PoolCreator;
  let owner: HardhatEthersSigner;

  before(async () => {
    const signers = await ethers.getSigners();
    owner = signers[0];
    tokenA = await new Token__factory().connect(owner).deploy('TokenA', 'To
    tokenB = await new Token__factory().connect(owner).deploy('TokenB', 'To
    poolCreator = await new PoolCreator__factory().connect(owner).deploy(UN
  });

  it('should create a pool', async () => {
    const contractAddress = await poolCreator.createPool.staticCall(tokenA,
    console.log('Contract Address', contractAddress);
    await poolCreator.createPool(tokenA, tokenB, 500);
  });
});
```

Notice the following:

> The address `0x1F98431c8aD98523631AE4a59f267346ea31F984` is the address of the Uniswap pool factory deployed to the Ethereum mainnet. This can be verified by looking at the Uniswap documentation that includes the **Deployment addresses of the contracts**.
>
> You created two tokens, TokenA and TokenB, by using a `Token` contract.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

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

contract Token is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbo
        _mint(msg.sender, 1000 ether);
    }
}
```

Finally, run `npx hardhat test` and you should get a result similar to the following:

```
PoolCreator tests
Contract Address 0xa76662f79A5bC06e459d0a841190C7a4e093b04d
    ✓ should create a pool (1284ms)

  1 passing (5s)
```

Interacting with external contracts using `.call()`

In the previous example, you accessed the Uniswap V3 Factory interface, however if you don't have access to the contract interface, you can use a special function called `call`.

Using `call`, you can call any contract as long as you know minimal information of

tokenB

fee

The newly modified smart contract code looks as follows:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

contract PoolCreator {
    address public uniswapFactory;

    constructor(address _factoryAddress) {
        uniswapFactory = _factoryAddress;
    }

    function createPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) external returns (address poolAddress) {
         bytes memory payload = abi.encodeWithSignature(
            "createPool(address,address,uint24)",
            tokenA,
            tokenB,
            fee
        );

        (bool success, bytes memory data) = uniswapFactory.call(payload);
        require(success, "Uniswap factory call failed");

        // The pool address should be returned as the first 32 bytes of the
        assembly {
            poolAddress := mload(add(data, 32))
        }
        require(poolAddress != address(0), "Pool creation failed");
        return poolAddress;
    }
}
```

Notice the following:

- By using `abi.encodeWithSignature`, you encode the payload required to make a smart contract call using the `.call()` function.
- Using `.call()` doesn't require you to import the interface.
- You load the pool address by using a special assembly operation called `mload`.

Try to run again the command `npx hardhat test` and you should expect the same result:

```
PoolCreator tests
Contract Address 0xa76662f79A5bC06e459d0a841190C7a4e093b04d
    ✓ should create a pool (1284ms)

  1 passing (5s)
```

Conclusion

Interfaces or the `.call` function are two ways to interact with external contracts. Using interfaces provides several advantages, including type safety, code readability, and compiler error checking. When interacting with well-documented contracts like Uniswap, using interfaces is often the preferred and safest approach.

On the other hand, the `.call` function offers more flexibility but comes with greater responsibility. It allows developers to call functions on contracts even without prior knowledge of their interfaces. However, it lacks the type safety and error checking provided by interfaces, making it more error-prone.

Was this page helpful? Yes No Suggest edits Raise issue

‹ **Testing the Interface** **Step by Step Guide** ›