Events

# Events

Copy page

Events in Solidity

In this article, you'll learn how events work in Solidity by reviewing some practical examples and common use cases of events.

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:

- Write and trigger an event
- List common uses of events

Understand events vs. smart contract storage

## Overview

Understanding how Solidity events work is important in the world of smart contract development. Events provide a powerful way to create event-driven applications on the blockchain. They allow you to notify external parties, such as off-chain applications, user interfaces, and any entity that wants to listen for events of a particular contract.

In this tutorial, you'll learn how to declare, trigger, and utilize events, gaining the knowledge necessary to enhance the functionality and user experience of your decentralized applications.

## What are events?

From the official solidity documentation, **events** are:

> *...an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.*

> *...when you call them, they cause the arguments to be stored in the transaction's log – a special data structure in the blockchain. These logs are associated with the address of the contract that emitted them, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of now, but this might change in the future).*

In other words, events are an abstraction that allow you to store a transaction's log information in the blockchain.

## Your first solidity event

Start by creating a first event in the `Lock.sol` contract that's included by default in Hardhat.

The event is called `Created` and includes the address of the creator and the amount that was sent during the creation of the smart contract. Then, `emit` the event in the constructor:

```
emit Created(msg.sender, msg.value);
```

The contract is:

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

contract Lock {
    uint public unlockTime;
    address payable public owner;

    event Created(address owner, uint amount);

    constructor(uint _unlockTime) payable {
        require(
            block.timestamp < _unlockTime,
            "Unlock time should be in the future"
        );

        unlockTime = _unlockTime;
        owner = payable(msg.sender);

        emit Created(msg.sender, msg.value);
    }
}
```

Events can be defined at the file level or as inheritable members of contracts (including interfaces). You can also define the event in an interface as:

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

interface ILock {
    event Created(address owner, uint amount);
}

contract Lock is ILock {
    uint public unlockTime;
    address payable public owner;

    constructor(uint _unlockTime) payable {
        require(
            block.timestamp < _unlockTime,
            "Unlock time should be in the future"
        );

        unlockTime = _unlockTime;
        owner = payable(msg.sender);

        emit Created(msg.sender, msg.value);
    }
}
```

You can test the event by simplifying the original test file with the following code:

```
import {
  time,
} from "@nomicfoundation/hardhat-toolbox/network-helpers";
import { ethers } from "hardhat";

describe("Lock tests", function () {
  describe("Deployment", function () {
    it("Should set the right unlockTime", async function () {
      const ONE_YEAR_IN_SECS = 365 * 24 * 60 * 60;
      const ONE_GWEI = 1_000_000_000;

      const lockedAmount = ONE_GWEI;
      const unlockTime = (await time.latest()) + ONE_YEAR_IN_SECS;

      // Contracts are deployed using the first signer/account by default
      const [owner] = await ethers.getSigners();

      // But we do it explicit by using the owner signer
      const LockFactory = await ethers.getContractFactory("Lock", owner);
      const lock = await LockFactory.deploy(unlockTime, { value: lockedAmou

      const hash = await lock.deploymentTransaction()?.hash
      const receipt = await ethers.provider.getTransactionReceipt(hash as s

      console.log("Sender Address", owner.address)
      console.log("Receipt.logs", receipt?.logs)

      const defaultDecoder = ethers.AbiCoder.defaultAbiCoder()
      const decodedData = defaultDecoder.decode(['address', 'uint256'], rec
      console.log("decodedData", decodedData)
    });
  });
});
```

Notice that the previous code is logging the sender address and the logs coming from the transaction receipt. You are also decoding the `receipts.logs[0].data` field that contains the information emitted by the event but not in a human-readable way, since it is encoded. For that reason, you can use `AbiCoder` to decode the raw data.

By running `npx hardhat test`, you should be able to see the following:

```
  Lock tests
    Deployment
Sender Address 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Receipt.logs [
  Log {
    provider: HardhatEthersProvider {
      _hardhatProvider: [LazyInitializationProviderAdapter],
      _networkName: 'hardhat',
      _blockListeners: [],
      _transactionHashListeners: Map(0) {},
      _eventListeners: []
    },
    transactionHash: '0xad4ff104036f23096ea5ed165bff1c3e1bc0f53e375080f84bc
    blockHash: '0xb2117cfd2aa8493a451670acb0ce14228b06d17bf545cd7efad6791ae
    blockNumber: 1,
    removed: undefined,
    address: '0x5FbDB2315678afecb367f032d93F642f64180aa3',
    data: '0x000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb922
    topics: [
      '0x0ce3610e89a4bb9ec9359763f99110ed52a4abaea0b62028a1637e242ca2768b'
    ],
    index: 0,
    transactionIndex: 0
  }
]
decodedData Result(2) [ '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266', 10000
      ✓ Should set the right unlockTime (1008ms)
```

Notice the value `f39fd6e51aad88f6f4ce6ab8827279cfffb92266` is encoded in the property data, which is the address of the sender.

## Event topics

Another important feature is that events can be indexed by adding the indexed attribute to the event declaration.

For example, if you modify the interface with:

```
interface ILock {
    event Created(address indexed owner, uint amount);
}
```

Then, if you run `npx hardhat test` again, an error may occur because the decoding assumes that the data field contains an `address` and a `uint256`. But by adding the indexed attribute, you are instructing that the events will be added to a special data structure known as “topics”. Topics have some limitations, since the maximum indexed attributes can be up to three parameters and a topic can only hold a single word (32 bytes).

You then need to modify the decoding line in the test file with the following:

```
const decodedData = defaultDecoder.decode(['uint256'], receipt?.logs[0].dat
```

Then, you should be able to see the receipt as:

```
 Lock tests
    Deployment
Sender Address 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Receipt.logs [
  Log {
    provider: HardhatEthersProvider {
      _hardhatProvider: [LazyInitializationProviderAdapter],
      _networkName: 'hardhat',
      _blockListeners: [],
      _transactionHashListeners: Map(0) {},
      _eventListeners: []
    },
    transactionHash: '0x0fd52fd72bca26879474d3e512fb812489111a6654473fd288c
    blockHash: '0x138f74df5637315099d31aedf5bf643cf95c2bb7ae923c21fcd7f0075
    blockNumber: 1,
    removed: undefined,
    address: '0x5FbDB2315678afecb367f032d93F642f64180aa3',
    data: '0x0000000000000000000000000000000000000000000000000000003b9aca
    topics: [
      '0x0ce3610e89a4bb9ec9359763f99110ed52a4abaea0b62028a1637e242ca2768b',
      '0x000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266'
    ],
    index: 0,
    transactionIndex: 0
  }
]
decodedData Result(1) [ 1000000000n ]
      ✓ Should set the right unlockTime (994ms)
```

Notice the topics property, which now contains the address of the sender: `f39fd6e51aad88f6f4ce6ab8827279cfffb92266` .

## Common uses of events

### User notifications

Events can be used to notify users or external systems about certain contract actions.

### Logging

Events are primarily used to log significant changes within the contract, providing a transparent and verifiable history of what has occurred.

### Historical state reconstruction

Events can be valuable for recreating the historical state of a contract. By capturing and analyzing emitted event logs, you can reconstruct past states, offering a transparent and auditable history of the contract's actions and changes.

## Debugging and monitoring

Events are essential for debugging and monitoring contract behavior, as they provide a way to observe what's happening on the blockchain.

The ability to use events to recreate historical states provides an important auditing and transparency feature, allowing users and external parties to verify the contract's history and actions. While not a common use, it's a powerful capability that can be particularly useful in certain contexts.

## Events vs. smart contract storage

Although it is possible to rely on events to fully recreate the state of a particular contract, there are a few other options to consider.

Existing services such as **The Graph** allow you to index and create GraphQL endpoints for your smart contracts and generate entities based on custom logic. However, you must pay for that service since you are adding an intermediate layer to your application. This has the following benefits, such as:

- the ability to simply query one particular endpoint to get all the information you need
- your users will pay less gas costs due to the minimization of storage usage in your contract

But storing all of the information within the smart contract and relying fully on it to access data can create more complexity, since not all of the data is directly query-able. The benefits of this approach include:

- your application requires only the smart contract address to access all of the required data
- there are fewer dependencies involved, which makes this approach more crypto native in the sense that everything is in the blockchain (but, storing all the data in the blockchain will cause higher gas costs)

As a smart contract developer, you must evaluate which options work best for you.

## Conclusion

In this lesson, you've learned the basics of Solidity events and their importance in Ethereum smart contract development. You now understand how to declare and trigger events, a few of their common use cases, and the difference between events and smart contract storage.

Now that you have a solid grasp of events and their versatile applications, you can leverage them to build more sophisticated and interactive smart contracts that meet your specific needs, all while being mindful of the cost considerations.

## See also

Was this page helpful? Yes No Suggest edits Raise issue

‹ Step by Step Guide Guide ›

base docs