LESSON SC 8 – Events

University of West Attica
Department of Electrical and Electronics Engineering
Ioannis Christidis
Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

In this lesson we will learn about `events` in blockchain.

In our SC, we created a function that a microcontroller can call to change the `status` of their car.

But how does the microcontroller know which car it is registered to?

Or how does it know which is the account that occupied the car so it will be unlocked only by it?

Or how does it know that the car is `OCCUPIED` and should change to `AVAILABLE`?

To inform entities outside of the blockchain about what is happening on-chain, we can use **`events`**.

Events

Events *in Solidity are a mechanism used to log information that is "emitted" during the execution of a SC and can be consumed by external applications, such as front-end interfaces or analytics tools.* Events are an essential part of interacting with the Ethereum blockchain because they enable communication between SCs and off-chain systems. *A transaction is required to trigger an event because events are part of the transaction log.* If you remember, back in *LESSON INTRO 4*, when we explained the transactions logs, there was a field called `logs`. The events we are going to create will be logged there.

In our case, we will use events to notify the microcontrollers of our DApp. We will create an event for each function that makes a transaction.

```
    event CarRegistered(uint256 indexed _carId, address indexed _mc);
```

Here, you can see how an `event` is declared. They must be declared inside the SC (similar to state variables). We tend to name the `events` like a result of the function where they are emitted (e.g. `function registerCar => event CarRegistered`.

`Events` *also take parameters.* These are important information about the function that our external applications can use (e.g. in the `CarRegistered` event, we emit the ID of the registered car, and the microcontroller `address` to notify the microcontroller about the car it is registered to).

indexed vs non-indexed

You will notice that in the parameters of the event there is an `indexed` keyword.

The `indexed` *keyword in Solidity is used in event parameters to enable filtering and efficient querying of events on the blockchain. You can mark up to three parameters of an event as indexed and this allows you to search for specific values in those parameters when retrieving event logs.*

Indexed parameters are stored in a special structure called `topics` *in the* `event log` which we will see later.

`Event` *parameters can be defined without* `indexed`*keyword, but they will be stored in the* `data section` *of the* `event log` *instead of the* `topics` *section. This means that they cannot be queried directly. To find logs with specific* `non-indexed` *values, you'd need to retrieve all logs and then manually process them off-chain.*

To `emit` an `event` in a function we can do the following:

```
    emit CarRegistered(_carId, _mc);
```

Logs

We will implement the `CarRegistered` event to our `registerCar` function and see the `logs`.

```
    function registerCar(address _mc, uint256 _price) public {
        // .... checks here (removed them for space in the slide)
        Car memory car = Car(_mc, _price, Status.UNAVAILABLE);
        uint256 currentCarId = nextCarId;
        cars[currentCarId] = car;
        nextCarId++;
        emit CarRegistered(currentCarId, _mc);
    }
```

Notice that we did not `emit` the price of the car on the `event`. This is not information that the microcontroller needs.

Now compile and deploy ICR and then use the `registerCar` function to register a new car. If you look at the `console` and find the `transaction` you can look the `logs`:

```
[
  {
    "from": "0xaE036c65C649172b43ef7156b009c6221B596B8b", // contract address
    "topic": "0x459e1abea8d58f1f390ad63904970bc184e7e026ff12f2f7f3c3aed6677ad187",
// The hash of the event's signature (CarRegistered(uint256,address)) (NOT important for now)
    "event": "CarRegistered", // emitted event
    "args": {
        "0": "0", // first parameter
        "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4", // second parameter
        "_carId": "0", // first parameter
        "_mc": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4" // second parameter
    }
  }
]
```

Rest of the `events`

From the `logs` we can understand that if we have :

1. the SC `address`
2. the `event` name
3. the expected `topics`

we can create off-chain logic that listens to the event when it is emitted.

For example, we can listen to `CarRegister event` from our DApp's microcontrollers to notify them about the car they are regstered to.

Now let's create events for all ICR functions that make transactions.

```
    event CarRented(uint256 indexed _carId, address indexed _mc, address indexed _user); // here we also emit the address of the
user that rented the car.
    event CarIsAvailable(uint256 indexed _carId, address indexed _mc);
    event CarIsUnavailable(uint256 indexed _carId, address indexed _mc);
```

Rest of the `events` (2)

~~In `rentCar` we will add the account that called the function in the event parameters.~~

```
function rentCar(uint256 _carId) public payable carIsValid(_carId) {
    // .... checks here (removed them for space in the slide)
    Car storage car = cars[_carId];
    car.status = Status.OCCUPIED;
    emit CarRented(_carId, car.mc, msg.sender);
}
```

Now let's do the same for `changeCarStatusMc`:

```
function changeCarStatusMc(uint256 _carId) public carIsValid(_carId) {
    // .... checks here (removed them for space in the slide)
    Car storage car = cars[_carId];
    car.status = Status.AVAILABLE;
    emit CarIsAvailable(_carId, car.mc);
}
```

And for `changeCarStatusOwner`:

```
function changeCarStatusOwner(uint256 _carId) public carIsValid(_carId) {
    // .... checks here (removed them for space in the slide)
    Car storage car = cars[_carId];
    if(car.status == Status.AVAILABLE) {
        car.status = Status.UNAVAILABLE;
        emit CarIsUnavailable(_carId, car.mc);
    } else {
        car.status = Status.AVAILABLE;
        emit CarIsAvailable(_carId, car.mc);
    }
}
```

Outro

This was a lot of information!

We learned how to inform microcontrollers about what happens inside the blockchain.

And we talked about:

`Events`

`Logs`

`Indexed parameters`

You see that our SC is getting big. Next lesson we are going to split it by using `inheritance`.