Blockchain and Distributed Ledger Technologies: Principles, Applications and Research Challenges Course Project

Politecnico di Milano

Giacomo Vinati and Matteo Savino

September 2023

What have we done?

- We **initially** focused on creating an **oracle smart contract**. As the development progressed, we aimed to embed a practical use case within it. Consequently, our project evolved to **include not just the information flow** from off-chain to on-chain, **but also** the comprehensive **logic to govern the entire application**. Specifically, our application revolved around orchestrating the ticketing and refund operations for a train company. It's worth noting that low effort would be required to adapt our system to other use cases (e.g., plane travels).
- Our **primary focus** was on enhancing the **Solidity section** since it's the central point of the project. We designed a **minimal CLI with Python** language to perform all the functions of the smart contract. As a matter of fact, our Python section is essentially a console application.
- Regarding the information exchange between blockchain and real world, we opted for the conventional **Inbound Push-Based pattern**. Off-chain entities initiate data transmission to the on-chain contract and the blockchain side emits events, which are listened by our system, in response.

Italy Trains - solidity code

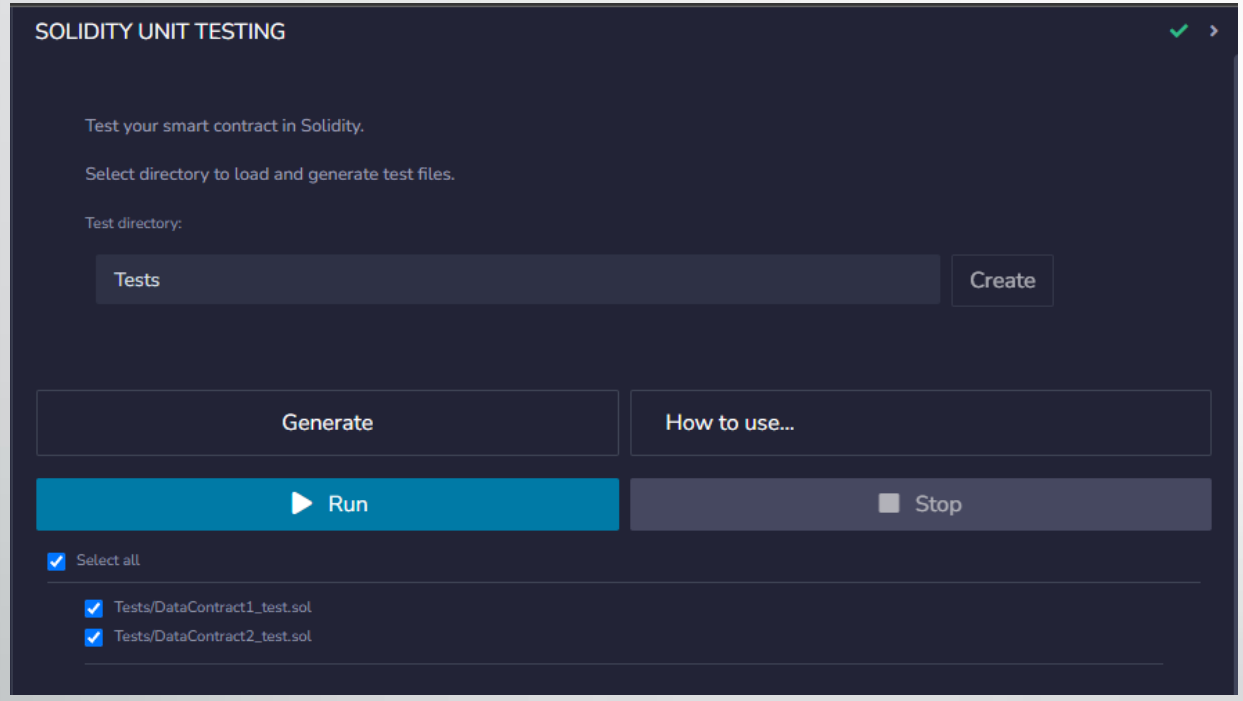
Smart Contract

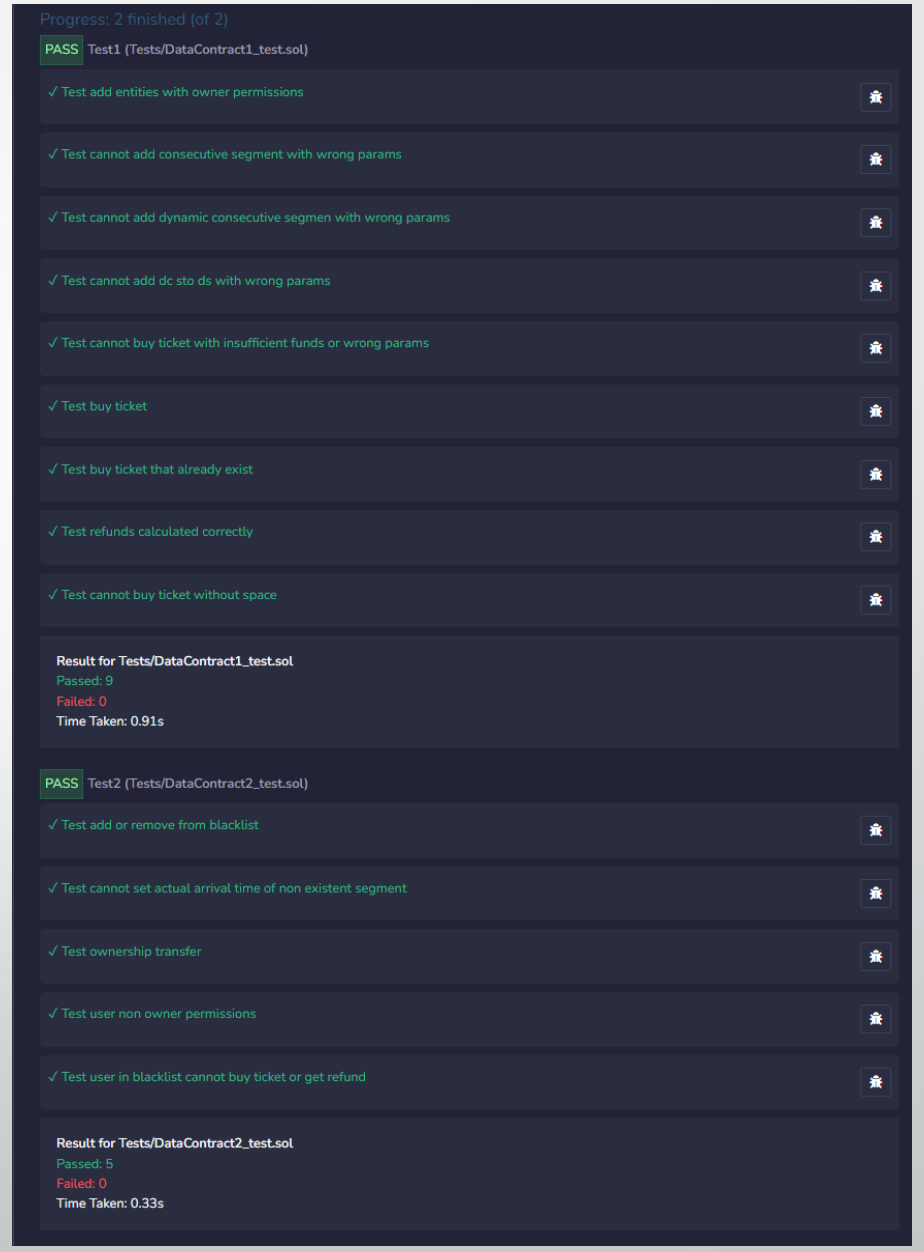
```
contract TrainsOracle {
    address public trainCompanyAddress;
    string public trainCompanyName = "ItalyTrains";

    mapping(address => bool) public blackList;

    struct Train {···
    }
    mapping(string => Train) public trains;

    mapping(string => bool) public stations;

    struct ConsecutiveSegment {···
    }
    mapping(string => ConsecutiveSegment) public consecutiveSegments;

    struct DynamicConsecutiveSegment {···
    }
    mapping(string => DynamicConsecutiveSegment)···
    //It maps to the consecutive segments ids all the dynamic segments
    mapping(string => string[])···
    struct DynamicSegment {···
    }
    mapping(string => DynamicSegment) public dynamicSegments;
    mapping(string => uint256) public dynamicSegmentPrices;

    struct DynamicTicket {···
    }
    mapping(string => DynamicTicket) public dynamicTickets;

    mapping(address => uint256) public refunds;

    modifier onlyOwner() {···
    }

    modifier notBlacklisted() {···
    }

    event RefundAdded(···
    );

    event RefundTaken(address passenger, uint256 refundAmount);

    constructor() payable {···
    }
```

```
receive() external payable {}

fallback() external payable {}

function setNewOwner(address newOwner) public onlyOwner {···
}

function addToBlacklist(address toBlackList) public onlyOwner {···
}

function removeFromBlacklist(address fromBlackList) public onlyOwner {
}

function addTrain(···
) public onlyOwner {···
}

function addStation(string calldata stationId) public onlyOwner {···
}

function addConsecutiveSegment(···
) public onlyOwner {···
}

function addDynamicConsecutiveSegment(···
) public onlyOwner {···
}

function addDynamicSegment(···
) public onlyOwner {···
}

function addDynamicConsecutiveSegmentToDynamicSegment(···
) public onlyOwner {···
}

function buyTicketStep(···
) internal notBlacklisted returns (uint256) {···
}

function buyDynamicTicket(···
) public payable notBlacklisted {···
}

function setArrivalTimeAndCheckRequiredRefunds(···
) public onlyOwner {···
}

function getRefund() public notBlacklisted {···
}
```

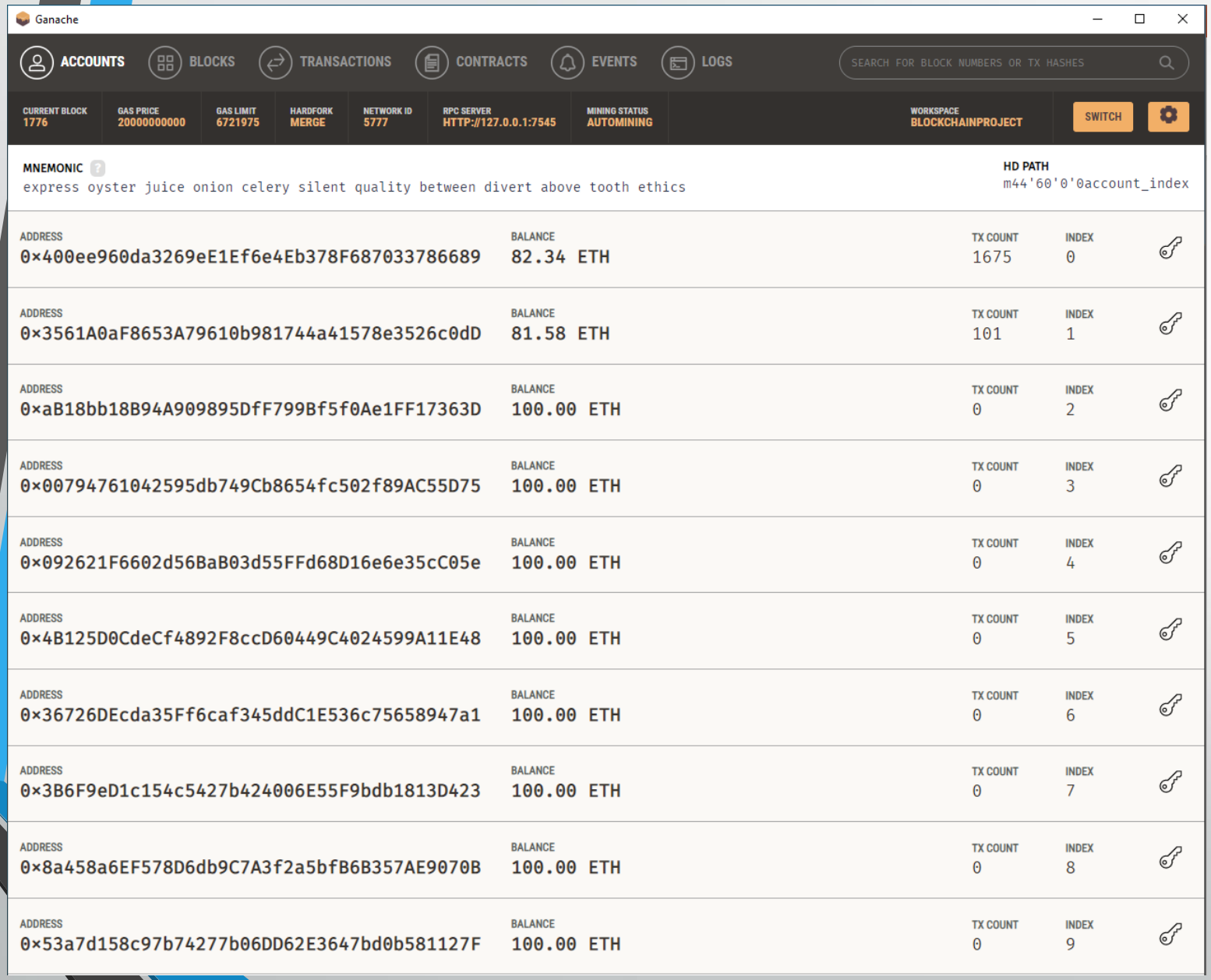
- Our **TrainsOracle** smart contract is designed to **automate the process of ticket purchases and refunds** for train travel companies. It allows to offload the responsibility of managing tickets and refunds to the smart contract, providing a seamless and transparent experience for customers.
- In order to make everything work correctly, we developed a **full model of the train company structure** including **trains, stations, segments**, etc.

Solidity Unit Testing



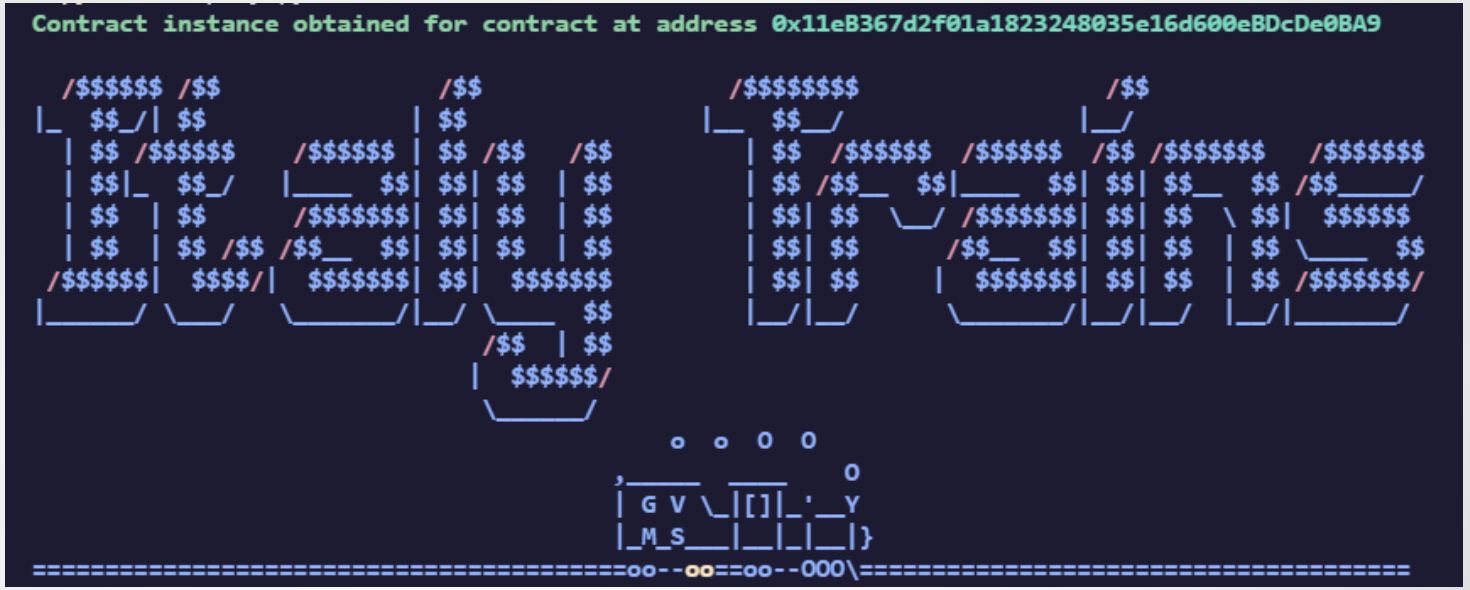


- We employed the **Solidity Unit Testing** extension within the *Remix IDE*.



Test RPC (Ganache)

In order **to simplify** the development, we opted to use **Ganache** which is a virtual environment which **simulates an entire blockchain** for development purposes.



Italy Trains - Python Cli

Compile and Deploy

- We create a **SmartContractUtility** class which contains some helpful functions that we use to compile and deploy the smart contract whenever necessary (at runtime).

```python
@staticmethod
def compile_contract(contract_source_path, contract_abi_path, contract_bytecode_path, contract_name):
    contract_source_code = SmartContractUtility.get_contract_source_code(
        contract_source_path)

    compiled_contract = compile_standard({
        "language": "Solidity",
        "sources": {
            contract_source_path: {
                "content": contract_source_code,
            },
        },
        "settings": {
            "outputSelection": {
                "*": {
                    "*": ["abi", "evm.bytecode"],
                },
            },
            "optimizer": {
                "enabled": True,
                "runs": 200      You, 3 weeks ago • Small fix …
            },
        },
    })

    contract_abi = compiled_contract["contracts"][contract_source_path][contract_name]["abi"]
    contract_bytecode = compiled_contract["contracts"][
        contract_source_path][contract_name]["evm"]["bytecode"]["object"]

    # Save ABI to the specified path
    with open(contract_abi_path, "w") as abi_file:
        json.dump(contract_abi, abi_file)

    # Save bytecode to the specified path
    with open(contract_bytecode_path, "w") as bytecode_file:
        bytecode_file.write(contract_bytecode)

    return contract_source_code, contract_abi, contract_bytecode
```

```python
@staticmethod
def deploy_contract(web3, contract_abi, contract_bytecode, sender_private_key, value=None, gas_limit=None, gas_price=None):
    sender_account = web3.eth.account.from_key(str(sender_private_key))
    nonce = web3.eth.get_transaction_count(sender_account.address)

    contract = web3.eth.contract(
        abi=contract_abi, bytecode=contract_bytecode)
    contract_constructor = contract.constructor()

    if gas_price is None:
        gas_price = web3.eth.gas_price

    if gas_limit is None:
        gas_limit = 3000000  # contract_constructor.estimate_gas()

    if value is None:
        value = web3.to_wei(1, 'ether')

    tx_params = {
        "nonce": nonce,
        "gas": gas_limit,
        "gasPrice": gas_price,
        "value": value,
        "from": sender_account.address
    }

    tx_hash = contract_constructor.transact(tx_params)
    tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)

    return tx_receipt.contractAddress
```

Company side

```
Welcome to Company CLI!

Please choose an action:
1. Add Train
2. Add Station
3. Add Consecutive Segment
4. Add Dynamic Consecutive Segment
5. Add Dynamic Segment
6. Add Dynamic Consecutive Segment To Dynamic Segment
7. Set Arrival Time And Check Required Refunds
8. Add User To Blacklist
9. Remove User From Blacklist
10. Set New Admin
11. Create Scenario
12. Exit
Enter your choice [1/2/3/4/5/6/7/8/9/10/11/12]:
```

Default scenario

- The **create_scenario** method generates a train **working scenario ready to use**. After execution, it displays the transaction outcome including all the details to the user.

```
DS1: S1 - S2 - S3 - S4 -- SlowTrain -- 0.04 ETH
DS2: S4 - S3 - S2 - S1 -- SlowTrain -- 0.04 ETH
DS3: S1 - S2 - S3 - S5 -- FastTrain -- 0.12 ETH
DS4: S5 - S3 - S2 - S1 -- FastTrain -- 0.12 ETH
DS5: S1 - S2 - S3 -- SlowTrain -- 0.03 ETH
DS6: S3 - S2 - S1 -- SlowTrain -- 0.03 ETH
DS7: S3 - S4 -- SlowTrain -- 0.01 ETH
DS8: S4 - S3 -- SlowTrain -- 0.01 ETH
DS9: S1 - S2 - S3 -- FastTrain -- 0.06 ETH
DS10: S3 - S2 - S1 -- FastTrain -- 0.06 ETH
DS11: S3 - S5 -- FastTrain -- 0.06 ETH
DS12: S5 - S3 -- FastTrain -- 0.06 ETH
User 0xC1F8c80A6E41C107a31F3C51FD57Ac25F4b7360a added to blacklist successfully!
Press ENTER to continue...
```

User side

```
Welcome to User CLI!
Current Balance of selected account is: 98.400428302 ETH

Please choose an action:
1. Buy Ticket
2. Collect Refunds Money
3. Exit
Enter your choice [1/2/3]: 1
```

Buy ticket

The **buy_ticket** method allows users to purchase train tickets. Users provide a unique ticket ID and select specific train segments they wish to book. They specify the payment amount in ETH. Before finalizing the purchase, a confirmation is sought. Once confirmed, the method interacts with the blockchain to complete the transaction, and the outcome (success or failure) is displayed to the user.

```
Welcome to User CLI!
Current Balance of selected account is: 98.400428302 ETH

Please choose an action:
1. Buy Ticket
2. Collect Refunds Money
3. Exit
Enter your choice [1/2/3]: 1
Enter an unique id for the ticket you want to buy: TK1
Enter a dynamic segment ID (or type 'done' to finish): DS1
Enter a dynamic segment ID (or type 'done' to finish): DS2
Enter a dynamic segment ID (or type 'done' to finish): done
Enter the value you want to pay for the ticket (in ETH) : 0.8
Are you sure you want to buy this ticket?  [yes/no]: yes
Ticket TK1 bought successfully! Paid fee: 0.009535856 ETH.
Updated Balance for 0xFa6B8901e5d805560868D7FEd7a54C7DA5bb0763: 97.591759334 ETH
Press ENTER to continue...
```

Set arrival time and check refunds

- Whenever a train reach a new station the actual arrival time must be communicated to the smart contact which will immediately check for necessary refunds.

```
Welcome to Company CLI!

Please choose an action:
1. Add Train
2. Add Station
3. Add Consecutive Segment
4. Add Dynamic Consecutive Segment
5. Add Dynamic Segment
6. Add Dynamic Consecutive Segment To Dynamic Segment
7. Set Arrival Time And Check Required Refunds
8. Add User To Blacklist
9. Remove User From Blacklist
10. Set New Admin
11. Create Scenario
12. Exit
Enter your choice [1/2/3/4/5/6/7/8/9/10/11/12]: 7
Enter the dynamic consecutive segment ID: DCS3
Enter the actual arrival timestamp: 2000000000
Are you sure you want to set this arrival time for that dynamic consecutive segment and check for refunds?  [yes/no]: yes
Arrival time 2000000000 set for dynamic consecutive segment DCS3 successfully and refund check completed!
Press ENTER to continue...
```

Events Listener Side

```
 python event.py
Listening for events from contract at address 0x11eB367d2f01a1823248035e16d600eBDcDe0BA9
Listening for RefundTaken events...
Listening for RefundAdded events...
Event Received: RefundAdded with data: AttributeDict({'passenger':
nger':
'0xFa6B8901e5d805560868D7FEd7a54C7DA5bb0763', 'dynamicSegmentId': 'DS1', 'amount':
40000000000000000, 'totalRefundToCollect': 40000000000000000})
Event Received: RefundAdded with data: AttributeDict({'passenger':
'0xFa6B8901e5d805560868D7FEd7a54C7DA5bb0763', 'dynamicSegmentId': 'DS1', 'amount':
nger':
'0xFa6B8901e5d805560868D7FEd7a54C7DA5bb0763', 'dynamicSegmentId': 'DS1', 'amount':
40000000000000000, 'totalRefundToCollect': 160000000000000000})
Event Received: RefundTaken with data: AttributeDict({'passenger':
'0xFa6B8901e5d805560868D7FEd7a54C7DA5bb0763', 'refundAmount': 160000000000000000})
```


THE END