Insper

Ativos Digitais e Blockchain

Ricardo Rocha
Raul Ikeda

Objetivo

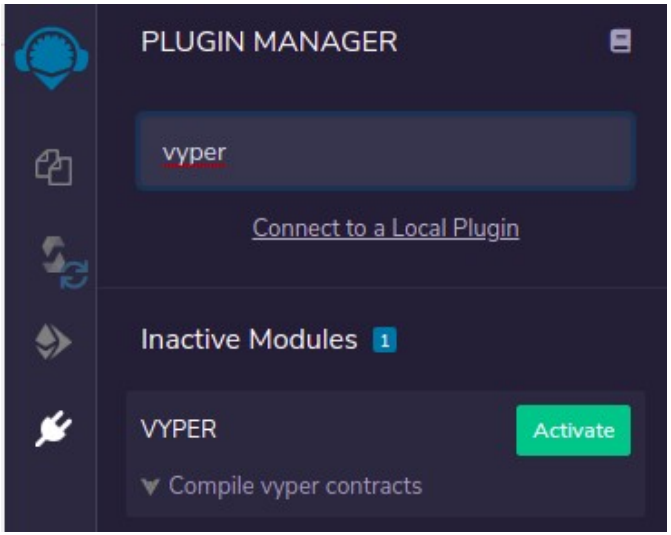
- Programação de Smart Contracts

Contexto

- Um consórcio de empresas solicitou a construção de uma solução para venda de ingressos
- Esses ingressos podem ser usados como passaporte de diversos eventos e portanto todos os locais precisam consultar se o ingresso é válido ou não para o portador
- O valor do ingresso pode ser modificado a qualquer momento
- Há um limite de número de ingressos
- Há uma cláusula de devolução/no show que permite a devolução de 80% do valor do ingresso

Solução

- Um Smart Contract que permite compra do ingresso ao cliente com cryptomoedas
- Esse contrato vai rodar em uma rede Ethereum e será desenvolvido em Vyper
- Utilizaremos o remix como plataforma de desenvolvimento
  - É preciso ativar o vyper como plugin



- Vamos fazer esse desenvolvimento desse contrato passo a passo cumprindo os requisitos do cliente

Insper

Requisito 1

“A aplicação deve permitir a compra do ingresso pelo cliente com cryptomoedas”

- O contrato deve ter uma função buy que recebe o endereço do comprador e o valor do ingresso e guarda a informação de que o comprador adquiriu um ticket

```
1  # Plataforma de ticket v0.0.1
2
3  # Dicionário que indica se o usuário comprou um ticket
4  # Se retornar 0, o usuário não possui ticket
5  users: public(HashMap[address, uint256])
6
7  @external # Habilita para interação externa (função chamável)
8  @payable # Habilita o recebimento de valores pela função
9  def buy():
10     # Preenche o dicionário com 1 no endereço de quem chamou a função
11     # msg.sender existe para toda função e não precisa entrar como argumento
12     self.users[msg.sender] = 1
```

- A função aceita qualquer valor de compra?

Requisito 1

- Vamos criar mais uma variável com o valor do ticket, essa variável não vai ser pública
- Para fazer o preenchimento do valor, vamos usar o construtor do contrato (função chamada na implantação)
- Ainda, vamos bloquear a execução da compra se o valor for inferior ao valor do ingresso (não damos troco)

```
# Plataforma de ticket v0.0.2

# Dicionário que indica se o usuário comprou um ticket
# Se retornar 0, o usuário não possui ticket
users: public(HashMap[address, uint256])
# Valor do ingresso
price: uint256

# Função que roda quando é feito o deploy do contrato
@external
def __init__(price: uint256):
    self.price = price

@external # Habilita para interação externa (função chamável)
@payable # Habilita o recebimento de valores pela função
def buy():
    # Testa se o valor passado ao contrato foi suficiente
    assert msg.value >= self.price

    # Preenche o dicionário com 1 no endereço de quem chamou a função
    # msg.sender existe para toda função e não precisa entrar como argumento
    self.users[msg.sender] = 1
```

Requisito 2

“O valor do ingresso pode ser modificado a qualquer tempo”

- Como declaramos a variável como privada (ou não pública) ela não é modificável diretamente.
- Vamos criar uma função change_price() com o novo valor de entrada

```
24  @external
25  def change_price(price: uint256):
26      self.price = price
27
```

- Problema: da forma como está feito, qualquer pessoa pode mudar o preço (não parece boa ideia).
- Solução: vamos bloquear para que apenas quem implantou o contrato possa modificar o preço

Requisito 2

Declaramos uma variável privada e alteramos o construtor para guardar o endereço do dono

```
 9  # Endereço do dono do contrato
10  owner: address
11
12  # Função que roda quando é feito o deploy do contrato
13  @external
14  def __init__(price: uint256):
15      # Guarda o Endereço do dono do contrato na variável
16      self.owner = msg.sender
17      self.price = price
```

- Ainda, alteramos a função change_price para permitir apenas que o dono modifique o preço

```
29  @external
30  def change_price(price: uint256):
31      assert msg.sender == self.owner
32      self.price = price
33
```

Requisito 3

“Há um limite para o número de ingressos emitidos”

- Usando a mesma ideia das soluções anteriores, vamos guardar um número máximo de tickets que vai ser passado na construção do contrato
- Criar um outra variável que vai contar o número de ingressos emitidos
- Na função de compra, verificar se o limite já não foi estourado.
  - Se ainda não foi, emitir e somar um no contador

- Executar as alterações: 15 minutos
- Usar o arquivo: Ticket_v4.vy

Requisito 3

```
12  # Número máximo de tickets
13  limit: uint256
14  # Número de tickets emitidos
15  count: uint256
16
17  # Função que roda quando é feito o deploy do contrato
18  @external
19  def __init__(price: uint256, limit: uint256):
20      # Guarda o Endereço do dono do contrato na variável
21      self.owner = msg.sender
22      self.price = price
23      self.limit = limit
24      self.count = 0
```

```
26  @external # Habilita para interação externa (função chamáv
27  @payable # Habilita o recebimento de valores pela função
28  def buy():
29      # Testa se o valor passado ao contrato foi suficiente
30      assert msg.value >= self.price
31
32      # Testa se ainda há tickets sobrando
33      assert self.count <= self.limit
34
35      # Preenche o dicionário com 1 no endereço de quem char
36      # msg.sender existe para toda função e não precisa ent
37      self.users[msg.sender] = 1
38
39      self.count += 1 # Soma 1 se finalizar a compra
```

Insper

BUG

“Permite o usuário comprar 2 vezes”

- Solução: vamos garantir que o comprar só pode comprar se o seu valor for zero no dicionário

```
26  @external # Habilita para interação externa (função chamável)
27  @payable # Habilita o recebimento de valores pela função
28  def buy():
29      # Testa se o comprador já não comprou
30      assert self.users[msg.sender] == 0
31
```

Requisito 4

“O usuário pode se arrepender, resgatando 80% do valor de volta”

- Criar uma função cancel()
- A função precisa verificar se:
  - O usuário comprou ingresso antes (valor == 1)
- Precisa anular o ingresso (valor = 0)
- Subtrair 1 ingresso do total emitido
- Transferir o valor de volta para o usuário

- Note que os testes estão começando a ficar complexos

Requisito 4

```
52  # A função não precisa ser payable
53  @external
54  def cancel():
55      # Testa se o comprador já comprou
56      assert self.users[msg.sender] == 1
57
58      # Anula a compra
59      self.users[msg.sender] = 0
60      # Subtrai 1 do contador de ingressos
61      self.count -= 1
62      # Devolve 80% do dinheiro (todos os valores tem que ser inteiros
63      send(msg.sender, self.price*80/100)
64
```

BUG

“Há uma cláusula de devolução/no show que permite a devolução de 80% do **valor do ingresso”**

O que acontece se o valor do ingresso subir durante a venda?

- Ideia:
  - Usar um outro Mapping para guardar o valor pago
  - Retornar o 80% do valor pago efetivamente
  - 15 minutos

Correção

```
# Plataforma de ticket v0.0.7b

# Dicionário que indica se o usuário comprou um ticket
# Se retornar 0, o usuário não possui ticket
users: public(HashMap[address, uint256])
# Valor do ingresso
price: uint256

# Guardar os preços pagos por cada um
prices: HashMap[address, uint256]
```

```
def buy():
    # Testa se o comprador já não comprou
    assert self.users[msg.sender] == 0

    # Testa se o valor passado ao contrato foi suficiente
    assert msg.value >= self.price

    # Testa se ainda há tickets sobrando
    assert self.count <= self.limit

    # Preenche o dicionário com 1 no endereço de quem chamou a função
    # msg.sender existe para toda função e não precisa entrar como argumento
    self.users[msg.sender] = 1

    # Guardar o preço real pago
    self.prices[msg.sender] = self.price
```

```
@external
def cancel():
    # Testa se o comprador já comprou
    assert self.users[msg.sender] == 1

    # Anula a compra
    self.users[msg.sender] = 0
    # Subtrai 1 do contador de ingressos
    self.count -= 1
    # Devolve 80% do dinheiro PAGO (todos os valores tem que ser inteiros)
    send(msg.sender, self.prices[msg.sender]*80/100)
```

Insper

Requisito 5

“Permitir a transferência de um ingresso para outra pessoa”

- Criar uma função transfer()
- A função precisa verificar se:
  - O endereço de quem chamou possui ingresso
  - O endereço de destino não possui ingresso
- Realizar a troca de ownership no dicionário

- Executar as alterações: 15 minutos
- Usar o arquivo: Ticket_v7.vy

Requisito 5

```
68  # Função que transfere um ingresso
69  @external
70  def transfer(destiny: address):
71
72      # Testa se o comprador já comprou
73      assert self.users[msg.sender] == 1
74
75      # Testa se o novo comprador não comprou
76      assert self.users[destiny] == 0
77
78      #transfere
79      self.users[msg.sender] = 0
80      self.users[destiny] = 1
```

Requisito 6

“Um botão para finalizar os eventos e sacar o dinheiro”

- Criar uma função end()
- A função precisa verificar se:
  - O endereço de quem chamou foi o próprio dono
- Sinalizar através de uma variável que o evento encerrou
  - Bloquear novas vendas e cancelamentos com essa variável
- Transferir o valor total do contrato para o dono

- Solução Final: Ticket_v9.vy

Próxima Aula

- Continuação
- Structs
- Tokenização