Distributed Algorithms II Project

Mauro Gentile & Peter Buttaroni

December 14, 2020

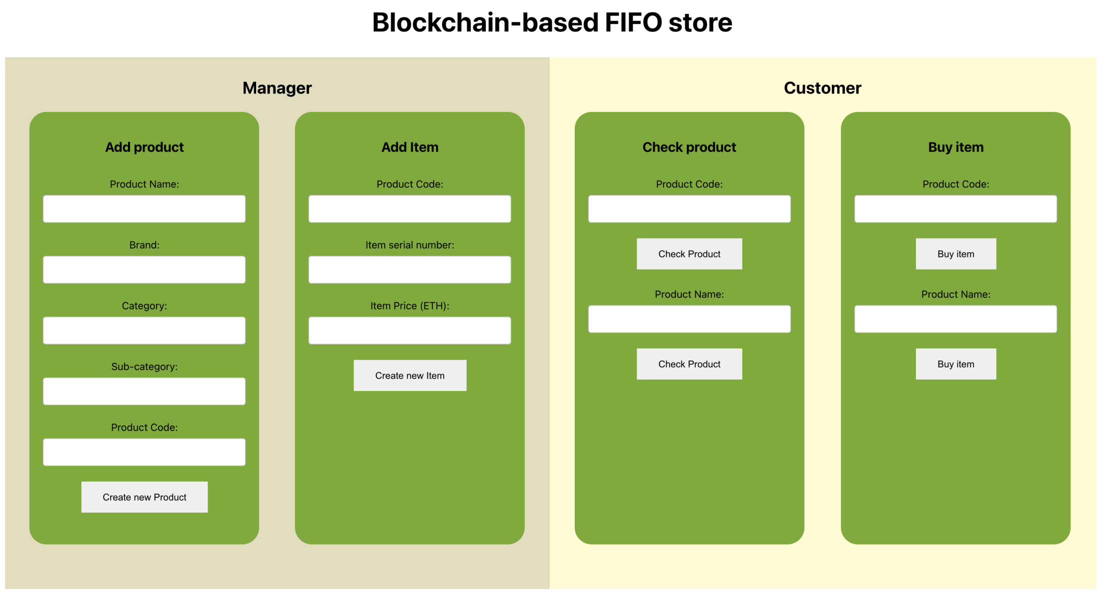
1 Introduction

1.1 Disclaimer

You can see the presentation at this link.
For the installation and to run the project, please refer to the **README** file in the attached folder.

1.2 Introduction to the project

The aim of this project is the implementation of a store's management on a distributed database based on blockchain. It also provides an intuitive **Web based GUI** for a friendly user experience.

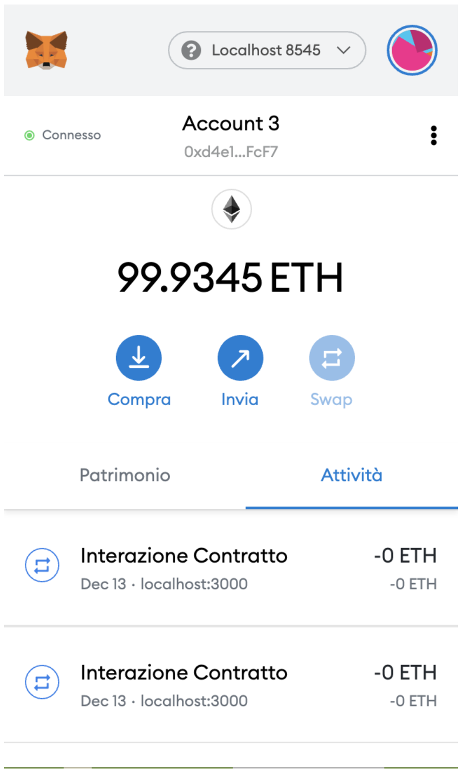


As you can see in the image above, the GUI is divided in two parts:

- The left part contains two operations: **Add product** and **Add Item**. These two function are available only for the owner of the store.
- The right part contains other two operations: **Check product** and **Buy item**. These functions are available for both the customers and the owner.

These two sections are meant to be separated in a real implementation (in particular, only the owner will have access to the GUI for the first two operations). Here they are shown together just for simplicity.

One important thing to mention is that almost all the possible errors and exceptions than may arise during the operations are managed through pop up window (high level) in order to guarantee a user friendly experience. All the operations are managed through this GUI and the interface of MetaMask that you can see in the next image.



99.9345 ETH

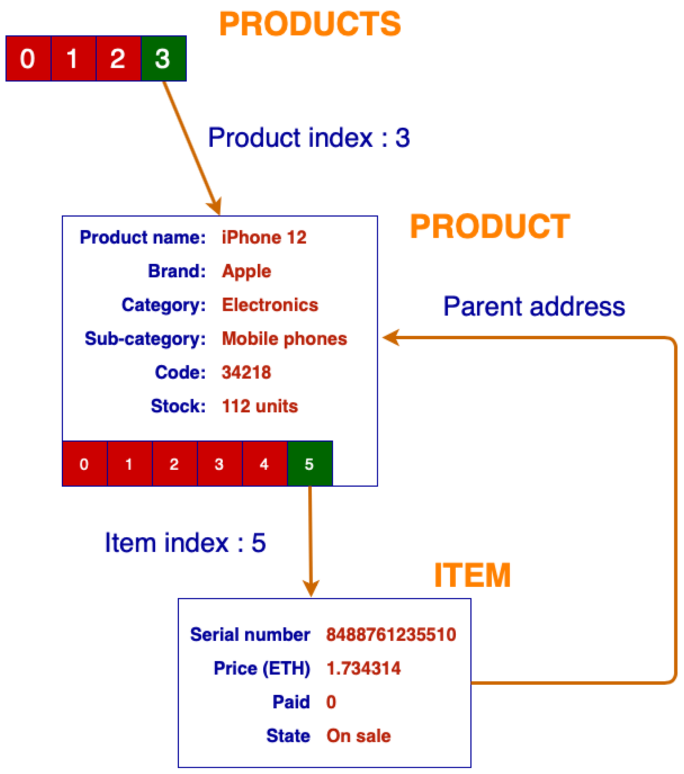
Interazione Contratto — Dec 13 · localhost:3000 — -0 ETH — -0 ETH

Interazione Contratto — Dec 13 · localhost:3000 — -0 ETH — -0 ETH

In the next sections we will first discuss about the data structures used in our project. Then, we will move toward the four available functions aforementioned.

2 Data Structures

Let's introduce the three main data structures that we have implemented for this project.
These Data structures are: **Products**, **Product**, and **Item**.
They are organized in a hierarchical way as you can see in the following image.



.

2.1 Products

The datastructure **Products** is the first element of our hierarchical scheme. It is a mapping of **Product**. The key is a progressive number that identifies in a unique way every single **Product**.

2.2 Product

The datastructure **Product** represent the abstraction of a single product (e.g. IPhone 12). It contains general information useful to both the customer and the manager:

- **Product name**: Is the name of the product (e.g. IPhone 12)
- **Brand**: It represents the name of the product's company (e.g. Apple)
- **Category**: It is the main category where the product belongs (e.g. Electronics).
- **Sub-category**: It is the secondary category where the product belongs (e.g. Mobile phones)
- **Code**: It is the number that identifies in a unique way the product (please notice that this is not the identifier of the product in the **Products** mapping).
- **Stock**: It represents the quantity of items of this product that the store has available for selling.
- **Product Index**: It is the identifier of the product in the **Products** mapping.
- **Mapping to Items**: Is a mapping that contains the references to all the items for a given product.

2.3 Item

The datastructure **Item** is the last element of our hierarchical scheme. It represents a single real item.
The **Item** contains useful information about a specific instance of the product.:

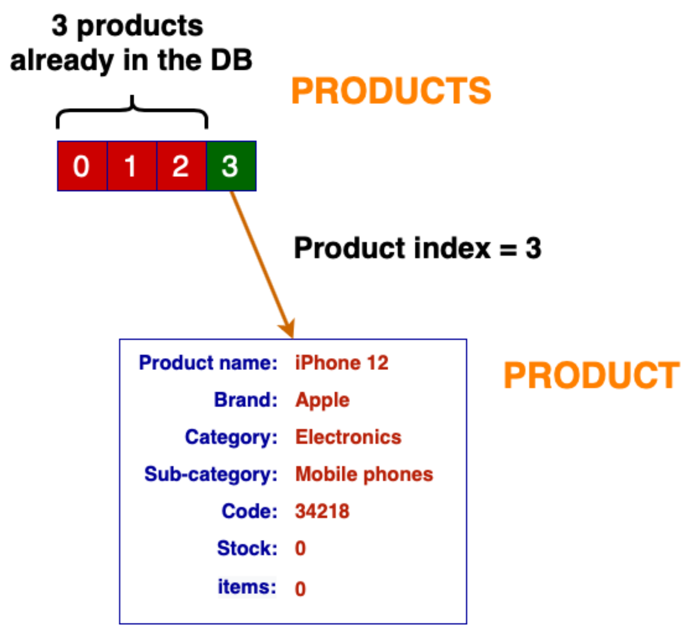
- **Serial Number**: It is a number that, given the **Product**, identifies in a unique way the item.
- **Price**: It is the price of the item expressed in Ether.
- **Paid**: This is a binary value that is 0 if the item has not been paid yet or 1 in the other case.
- **State**: It represents the current state of the item. This value can be: **OnSale**, **Paid**.

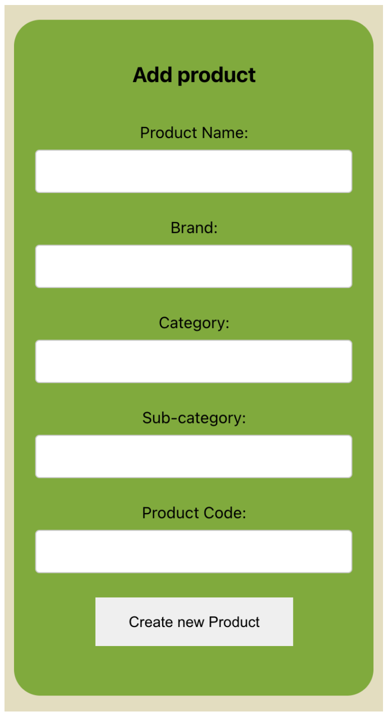
Moreover, every Item is characterised by its own address. This address will be used when a customer wants to buy that item. The money will be send to that specifc address and the customer will become the owner of that item.

3 Four operations

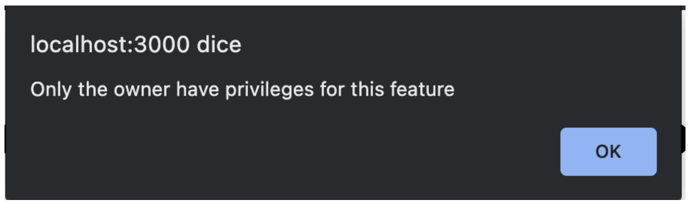
3.1 Add product

The **Add Product** function allows to the owner to add a new product to his own store.
When we add a product through the relative operation on the GUI, we creates a new element of the mapping **Products**. The following image on the left represents the scheme of the operation performed. On the right, we have the relative GUI section.





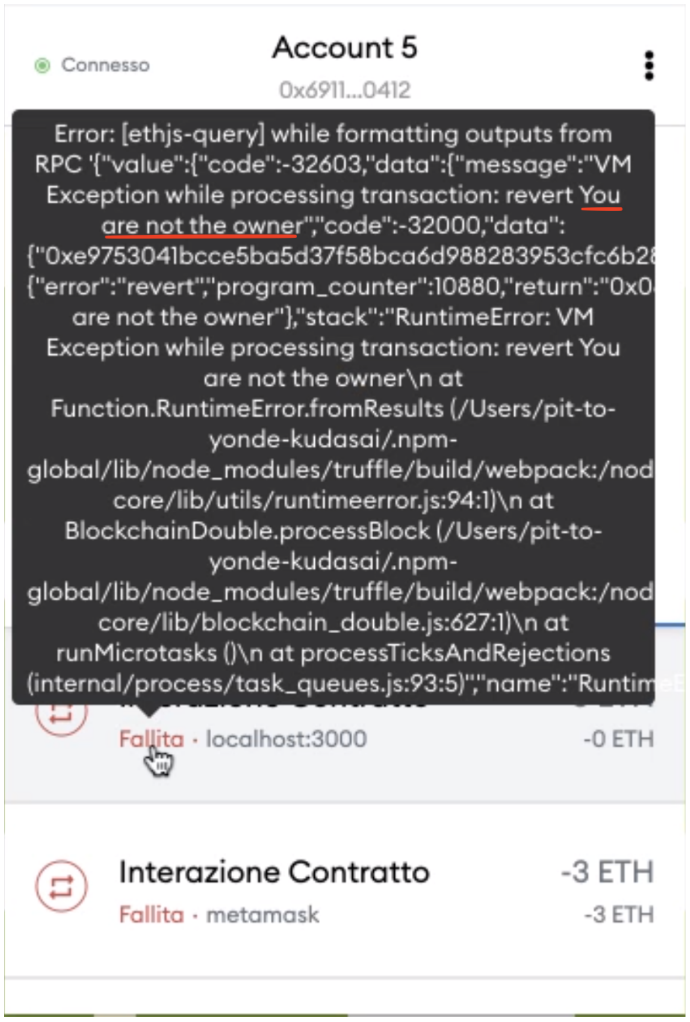
As you can see, the product is added to the first free slot. It contains different useful information for both the customer and the owner. Most of these information (Product name, Brand, Category, etc.) are provided by the owner when he fills in the fields of the GUI. Other (Stock, Items mapping, etc.) are created automatically by the code.
Is important to highlight that, before any real operation, we check that the address of the function caller is the same of the owner. If this is not the case, the code raises an error and stops (as you can see in the next image).



localhost:3000 dice

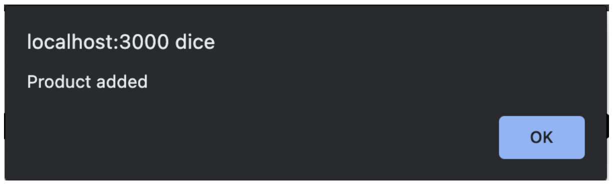
Only the owner have privileges for this feature

We can see this error also from the log in the MetaMask account.



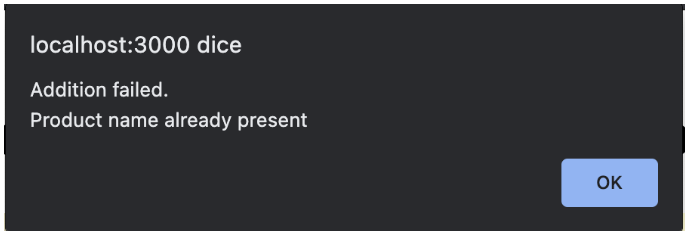
```
Error: [ethjs-query] while formatting outputs from
RPC '{"value":{"code":-32603,"data":{"message":"VM
Exception while processing transaction: revert You
are not the owner","code":-32000,"data":
{"0xe9753041bcce5ba5d37f58bca6d988283953cfc6b2
{"error":"revert","program_counter":10880,"return":"0x0
are not the owner"},"stack":"RuntimeError: VM
Exception while processing transaction: revert You
are not the owner\n at
Function.RuntimeError.fromResults (/Users/pit-to-
yonde-kudasai/.npm-
global/lib/node_modules/truffle/build/webpack:/nod
core/lib/utils/runtimeerror.js:94:1)\n at
BlockchainDouble.processBlock (/Users/pit-to-
yonde-kudasai/.npm-
global/lib/node_modules/truffle/build/webpack:/nod
core/lib/blockchain_double.js:627:1)\n at
runMicrotasks ()\n at processTicksAndRejections
(internal/process/task_queues.js:93:5)","name":"Runtim
```

On the other hand, if address is the same of the owner and the operation succeds, we will get a pop up window as the following one:

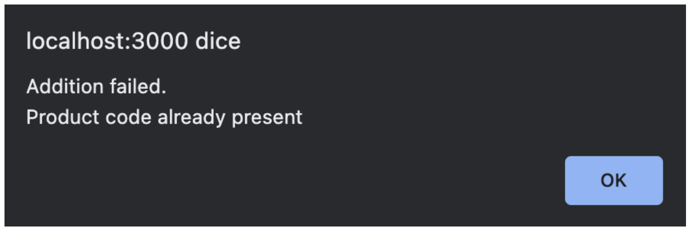

localhost:3000 dice

Product added

If we try to add a new product with the same name or the same code, we will get an error as the following ones (the first is in the case of same name, the second in the case of the same code). In this case, no amount of gas will be paid.


localhost:3000 dice

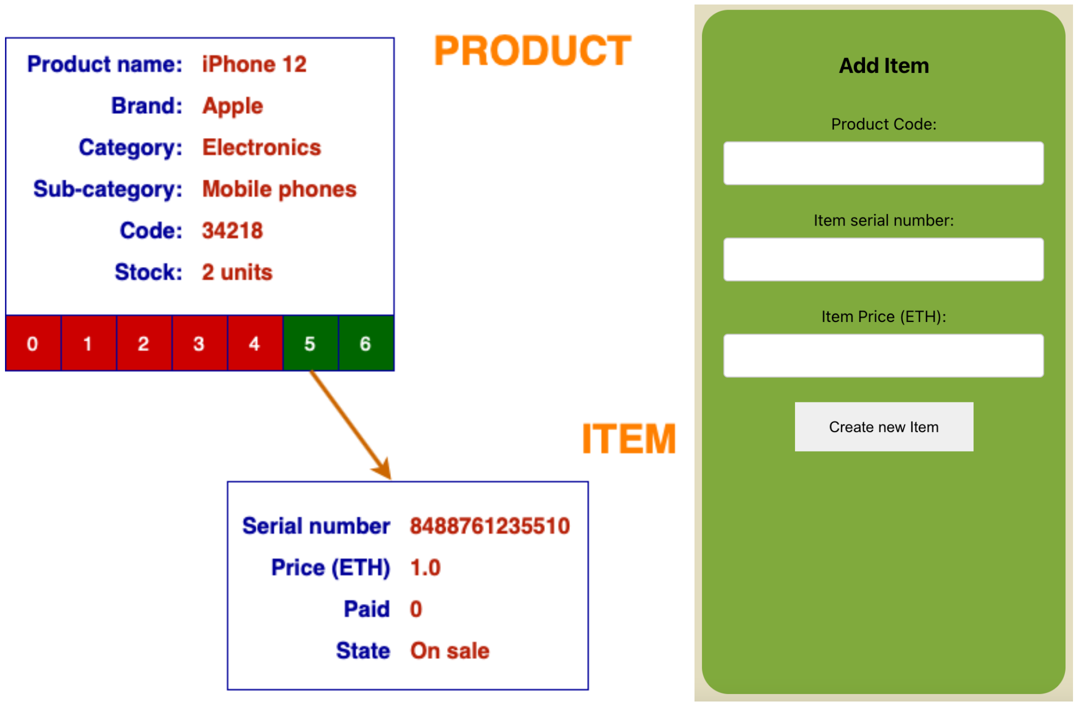
Addition failed.
Product name already present


localhost:3000 dice

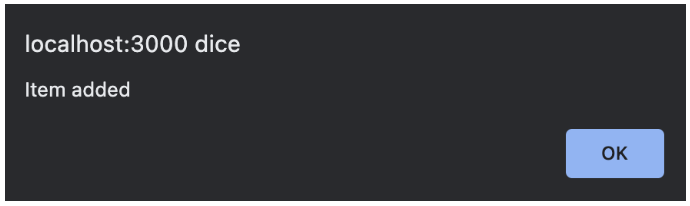
Addition failed.
Product code already present

3.2 Add Item

The **Add Item** function is the second function available exclusively for the owner. It allows you to add an item for a specific product.
The following image represents on the left the scheme of the operation and on the right the relative section in the GUI.

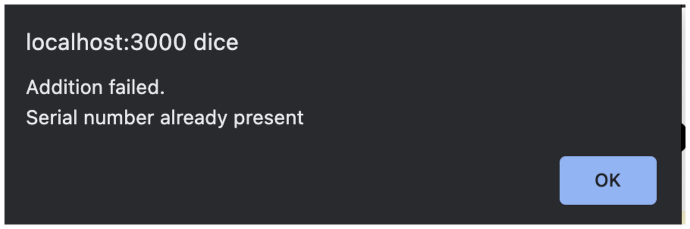


Also in this case, the new Item is added to the first available slot in the mapping inside the product. When we create a new Item, we need to specify the product at which it belongs, the serial number of the specific item (that uniquely identifies an item), and the price. At the beginning, the state of the item is 'On sale', this means that is ready to be sold. As we will see, this state becomes 'Paid' once a user buys the item.
When we call **Add Item**, the code checks the identity of the caller. If this coincides with the address of the owner we will be able to proceed, otherwise we will get an error.
If the operation succeeds, we will get the following pop up window.


localhost:3000 dice

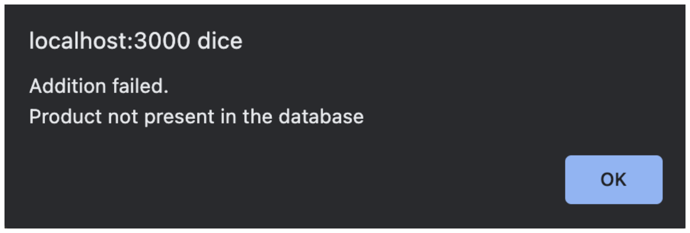
Item added

If we succeed to add an item, the stock number of the relative product will increase by one. For a given product, if we try to add an item with a serial number that already exists for that product, we will get the following errors.


localhost:3000 dice

Addition failed.
Serial number already present

On the other hand, if we try to add an item for a product that does not exist, it will be raised the following error.

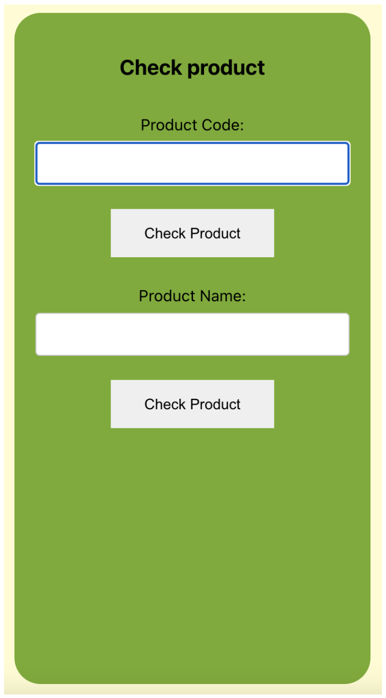

localhost:3000 dice

Addition failed.
Product not present in the database

In all these error cases, there will be no gas consumption.

3.3 Check Product

The **Check Product** is the first function in the area available for both the customer and the owner. This function allows the user to check the existance and the information of a given product. The search can be made both on product name or product code (as you can see in the following images).

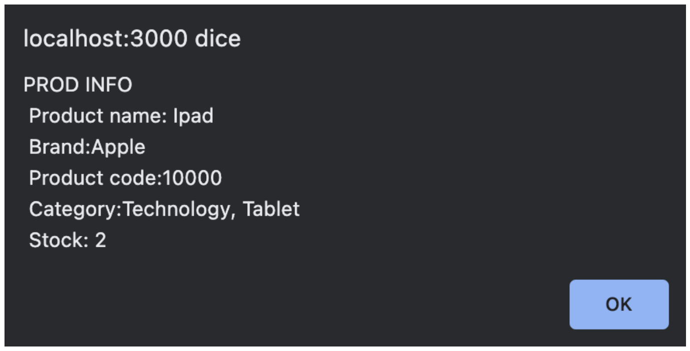


This operation does not modify the smart contract. Therefore, it can be performed without any consumption of gas. To do this, we use a view function as the following one.

```
function gatherProductInfo_by_code(uint _code) public view
returns(string memory identifier, uint code, uint stock, string memory brand, string memory category, string memory subCategory, bool outcome)
```

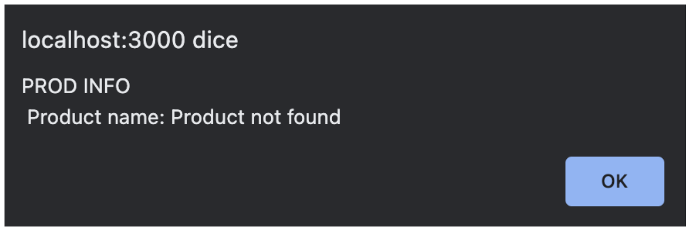
In this way, when we call the function on the local node, we do not consume gas.

If we are looking for an existing product, after the call of the function, we will get a pop up window similar to the next one.



```
localhost:3000 dice
PROD INFO
 Product name: Ipad
 Brand:Apple
 Product code:10000
 Category:Technology, Tablet
 Stock: 2
```

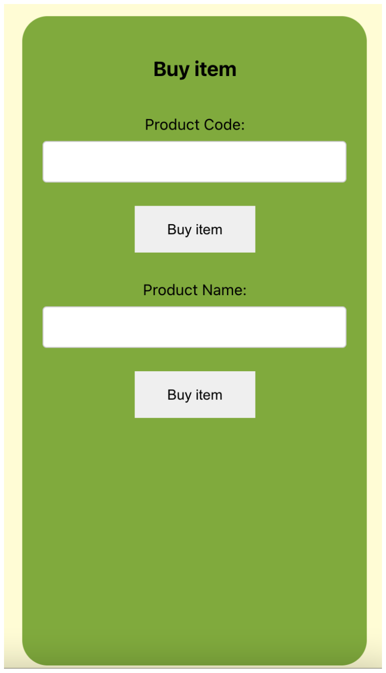
As we can see, it will be showed the name and the product code. Moreover we will see the brand, the category, and subcategory. Eventually we can see the stock. This is the number of Product's items available for sale at the moment.
If the product we are looking for does not exist, we will see the following warning.



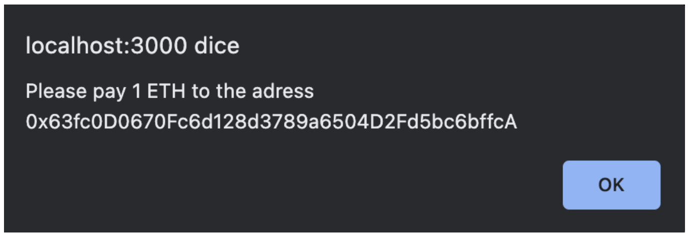
```
localhost:3000 dice
PROD INFO
 Product name: Product not found
```

3.4 Buy Item

The last operation we will considered is **Buy Item**. This function allows to the user to buy an item of the product he is looking for.



The picture above shows the section of the GUI dedicated to this operation. Also in this case, the user can search a product using either the name or the code.
If the product we are looking for exists and has at least one item available, we will get the following pop up window as return of our search.



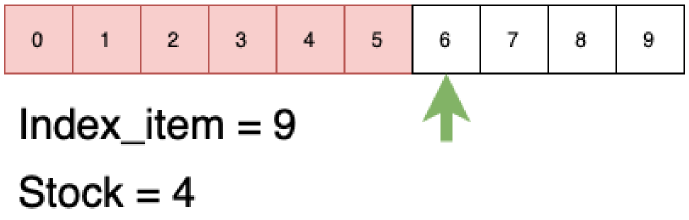
localhost:3000 dice

Please pay 1 ETH to the adress
0x63fc0D0670Fc6d128d3789a6504D2Fd5bc6bffcA

Specifically, the address returned is the one of the first available item in the mapping. The position of this item is found using the following rule:

$$index = index\_item - stock + 1$$

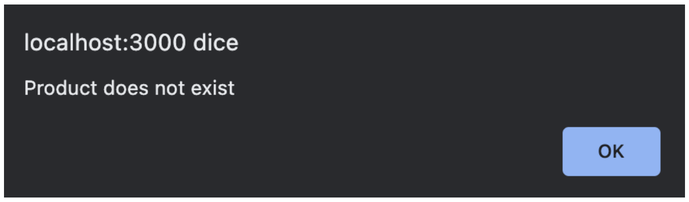
Let's make a quick example. In the next image, we have an $Index\_item = 9$ and $Stock = 4$



This means that the first available will be:

$$index = index\_item - stock + 1 = 9 - 4 + 1 = 6$$

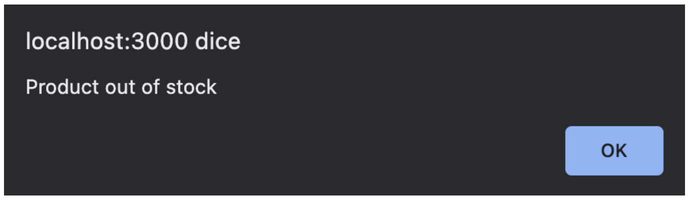
Now, let's go back to the discussion about the purchasing.
If the product we are looking for does not exist, we will see the following error.



localhost:3000 dice

Product does not exist

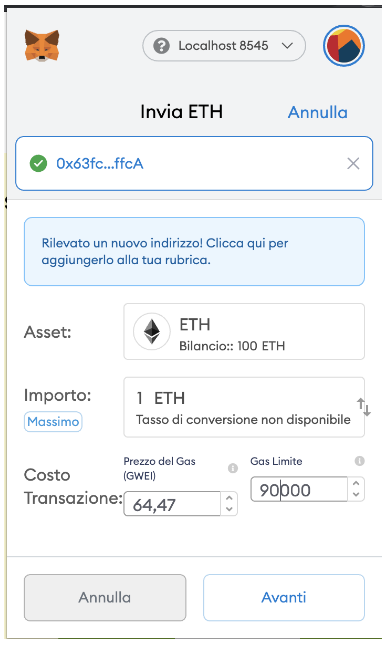
Eventually, if the product exists but there are no items available, we will get the following pop up window.



localhost:3000 dice

Product out of stock

Once we get the amount that we have to pay and the item address, we can use metamask to complete the transaction. Therefore, we select one account, we go on the 'send' section and we paste the item address.


Localhost 8545

Invia ETH Annulla

0x63fc...ffcA

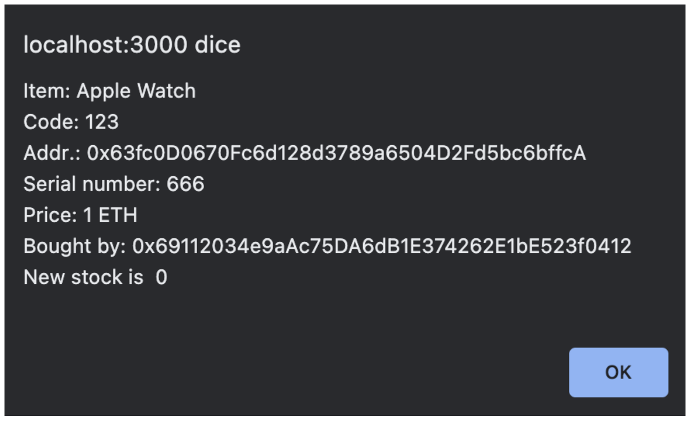
Rilevato un nuovo indirizzo! Clicca qui per aggiungerlo alla tua rubrica.

Asset: ETH
Bilancio:: 100 ETH

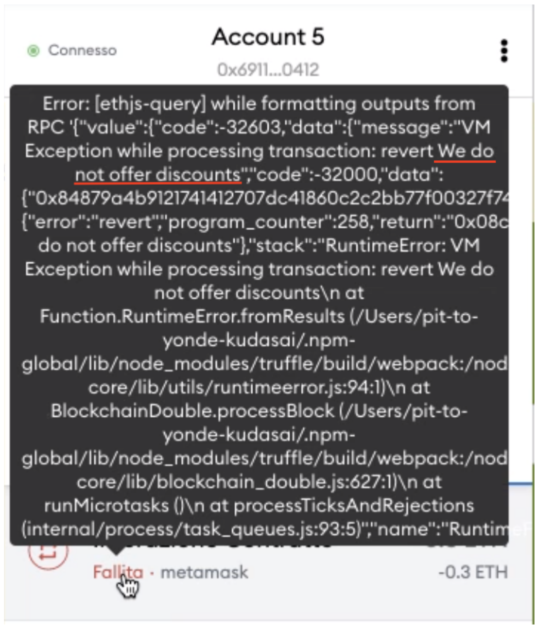
Importo: 1 ETH
Tasso di conversione non disponibile

Costo Transazione:
Prezzo del Gas (GWEI): 64,47
Gas Limite: 90000

After submitting the transaction, it will be processed and, if everything will be fine, at the end we will get a pop up window similar to the following one.


localhost:3000 dice
Item: Apple Watch
Code: 123
Addr.: 0x63fc0D0670Fc6d128d3789a6504D2Fd5bc6bffcA
Serial number: 666
Price: 1 ETH
Bought by: 0x69112034e9aAc75DA6dB1E374262E1bE523f0412
New stock is 0

In this case, the stock for that product will decreases by one (since we have one item less). If the customer sends few money or too much, it will appear one of the following alerts in the log of the MetaMask account (on the left the alert in case we are sending not enough money, on the right the alert in the opposite case).

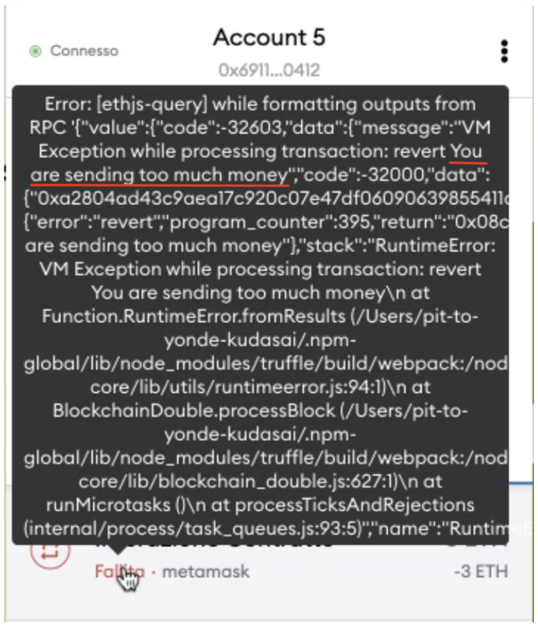

Connesso

Account 5

0x6911...0412

```
Error: [ethjs-query] while formatting outputs from
RPC '{"value":{"code":-32603,"data":{"message":"VM
Exception while processing transaction: revert We do
not offer discounts","code":-32000,"data":
{"0x84879a4b9121741412707dc41860c2c2bb77f00327f74
{"error":"revert","program_counter":258,"return":"0x08c
do not offer discounts"},"stack":"RuntimeError: VM
Exception while processing transaction: revert We do
not offer discounts\n at
Function.RuntimeError.fromResults (/Users/pit-to-
yonde-kudasai/.npm-
global/lib/node_modules/truffle/build/webpack:/nod
core/lib/utils/runtimeerror.js:94:1)\n at
BlockchainDouble.processBlock (/Users/pit-to-
yonde-kudasai/.npm-
global/lib/node_modules/truffle/build/webpack:/nod
core/lib/blockchain_double.js:627:1)\n at
runMicrotasks ()\n at processTicksAndRejections
(internal/process/task_queues.js:93:5)","name":"Runtim
```

Fallita · metamask -0.3 ETH


Connesso

Account 5

0x6911...0412

```
Error: [ethjs-query] while formatting outputs from
RPC '{"value":{"code":-32603,"data":{"message":"VM
Exception while processing transaction: revert You
are sending too much money","code":-32000,"data":
{"0xa2804ad43c9aea17c920c07e47df06090639855411c
{"error":"revert","program_counter":395,"return":"0x08c
are sending too much money"},"stack":"RuntimeError:
VM Exception while processing transaction: revert
You are sending too much money\n at
Function.RuntimeError.fromResults (/Users/pit-to-
yonde-kudasai/.npm-
global/lib/node_modules/truffle/build/webpack:/nod
core/lib/utils/runtimeerror.js:94:1)\n at
BlockchainDouble.processBlock (/Users/pit-to-
yonde-kudasai/.npm-
global/lib/node_modules/truffle/build/webpack:/nod
core/lib/blockchain_double.js:627:1)\n at
runMicrotasks ()\n at processTicksAndRejections
(internal/process/task_queues.js:93:5)","name":"Runtim
```

Fallita · metamask -3 ETH