Restaurant Dapp Report

For the owner, each restaurant's and each customer's account, I have first opened Ganache. From Ganache, I copy the private key and load it inside Metamask. Then, after inserting how many accounts I wanted, I renamed them and then used the option in Metamask: localhost 8545. It gives me 100 test Ether for every account. All the examples below need no more than 5 accounts: Owner, Restaurant 1, Restaurant 2, Owner 1, Owner 2. Each time we want to trigger a function with a specific one of the above accounts, we need to swap to that first in Metamask before pressing any button to trigger a function.

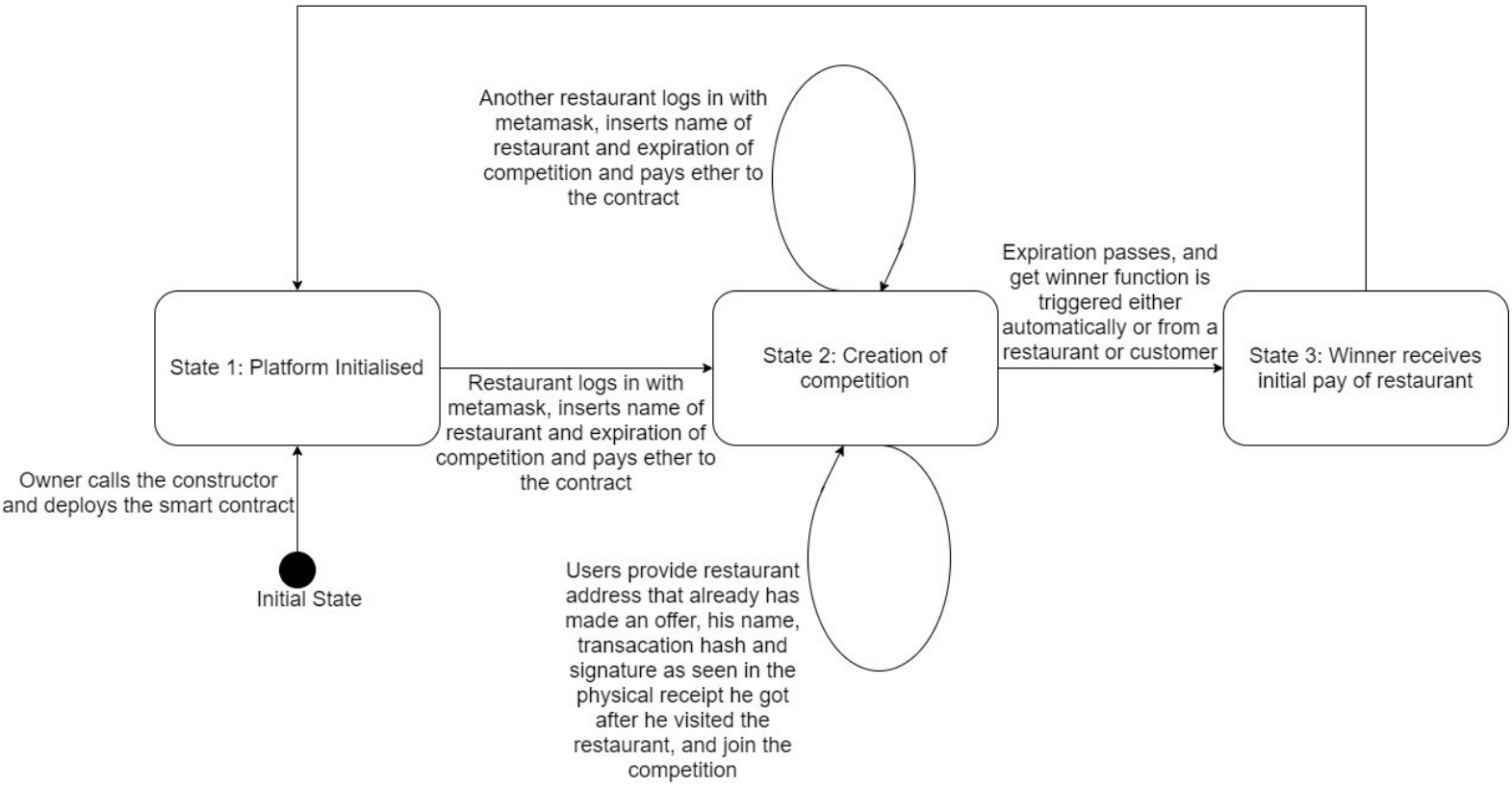
I have created a user-friendly Dapp, that features restaurants that want to advertise as well as reward their customers. First, an owner calls the constructor of the solidity contract and the contract is deployed on the blockchain. He needs first to have an account in Metamask and just run the server that will do the job (navigate to the server location and do "python server.py"). Note that the owner is not an administrator or anything, he is just deploying the contract, and cannot interfere with anything in the smart contract as an authoritative group could do.

Then, any restaurant that has a Metamask address can create a competition for its future customers. It can do that by loading the intro web page(http://127.0.0.1:5000/intro) and pressing that it is a restaurant that will redirect it to the restaurant platform. The restaurant writes down its name and denotes in seconds how much time for the expiration of joining the competition from customers in seconds. After that, it presses to create the competition and pays the amount of Ether to the contract which will be the amount that the winner will win (for now, it always pays 1 Ether, but it would be simple to have an extra field for amount). Many competitions can be going on from different restaurants at the same time. However, a single restaurant cannot have more than one ongoing competition at the same time.

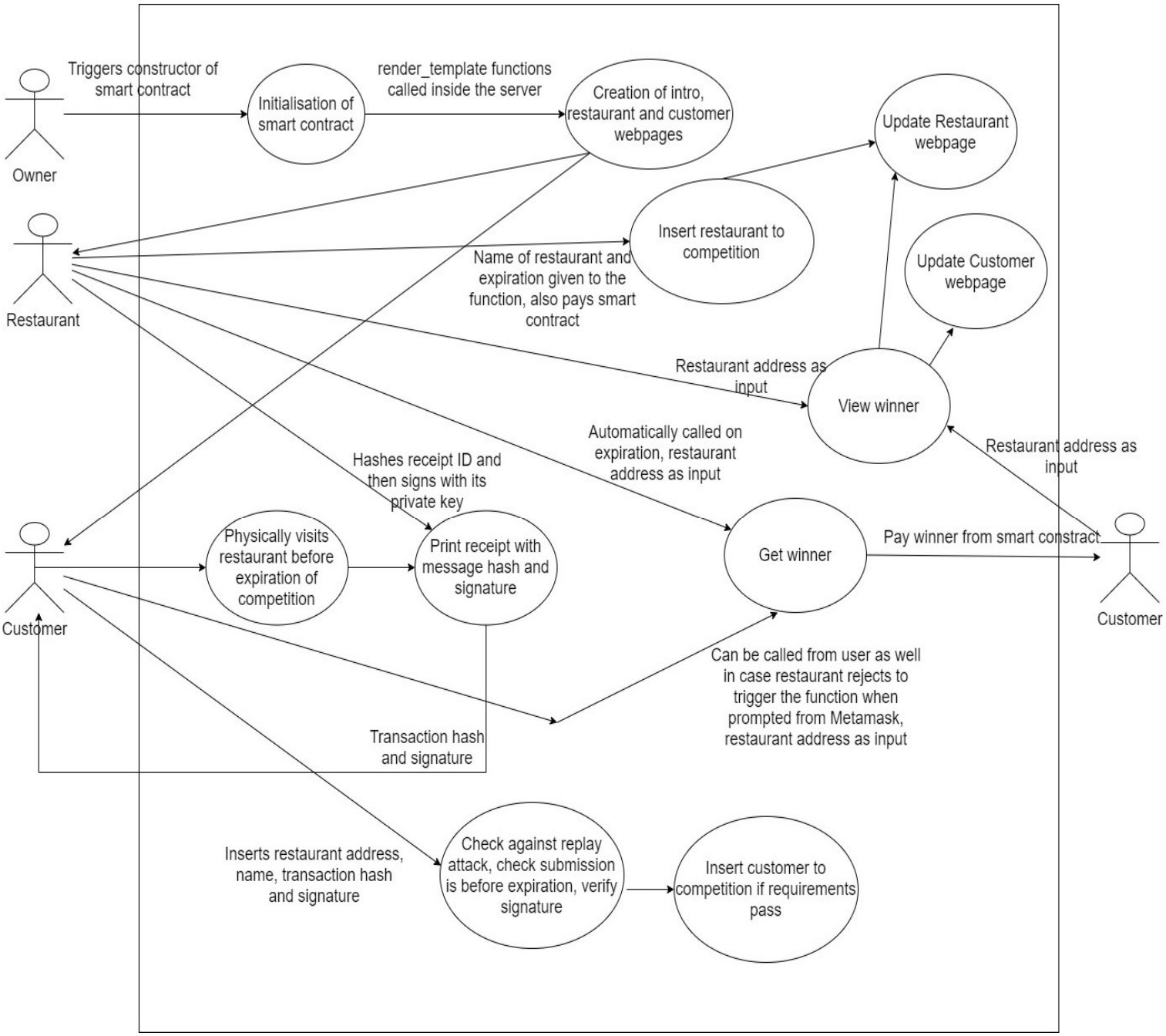
When a competition is ongoing, each customer that physically visits the restaurant before the expiration will have the chance to join and potentially win. In the receipt, the restaurant will write down for the customer the transaction hash of the receipt ID as well as sign that hash and gives the signature to the customer. The restaurant can do that via the restaurant platform: It just inserts in the signature section the receipt ID, and then clicks sign getting the hashed message and the signature. Moreover, it provides to the customer the public address of the restaurant. The customer needs to have a Metamask account as well in order to join the competition. It opens the intro page as well and clicks that it is a customer, redirecting him in the customer platform. From there, he/she inserts the address of the restaurant, his/her name, the hashed message and the signature. The solidity contract performs a series of checks that need to pass for the customer to join the competition. These are: the expiration should not have passed, we check against a replay attack – If the customer has already joined the competition, we do not let him join again, we do not let restaurants that have ongoing competitions join any competition as customers, and, lastly, we verify in the solidity contract that the signature of the hashed message was performed from the key of the restaurant, otherwise the customer will not be able to join.

Once the expiration has passed, either a restaurant or a user can press the button “Get winner” that triggers a function in the solidity contract and randomly assigns a winner from all the participants. I have implemented a function in JavaScript, that on expiration, the active Metamask account gets the address that is specified in the field of which restaurant to find the winner and calls get winner on that. So, a correct restaurant is assumed to provide its address on that field before the expiration so that when the exact time comes, it triggers that function and we get a winner. However, the restaurant may choose not to do that, either intentionally or accidentally. So, the Get winner button is both on the restaurant and customer platform and can be called by anyone. Note that there is a requirement in the solidity contract that there should be no previous winner – this makes sure we only trigger this function once. Also, it makes sure that the expiration has passed. In order to get a random winner, we perform the keccak256 of the current time and the current block timestamp that is only in the control of the miner and not on the restaurant or any customer and then we do a modulo of how many customers joined the competition. The winner that is found from there automatically receives the Ether from the contract to its account. There is also a “Show winner” button in both the webpages that is a view function and do not cost and gas to call. This just provides the address that won the competition and received the ether. If there is no winner yet, it just shows the zero address.

Below, you can see a complete State Diagram of the Dapp:



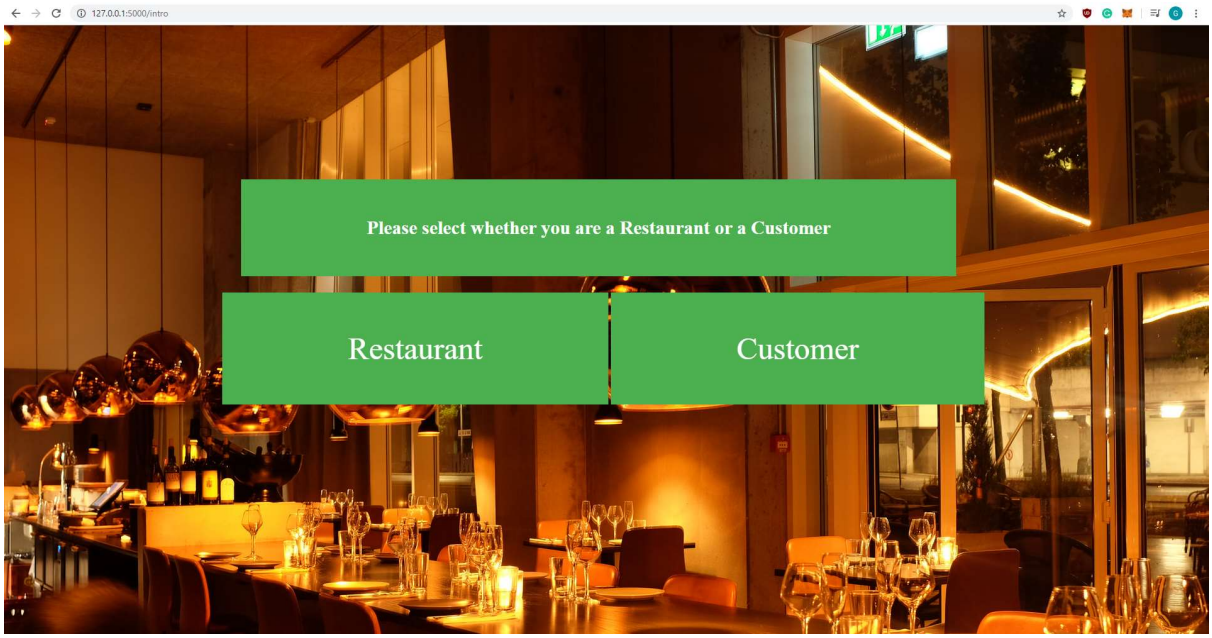
Also, below is a detailed UML diagram:



Demonstration of use-cases with screenshots and explanation from the UML:

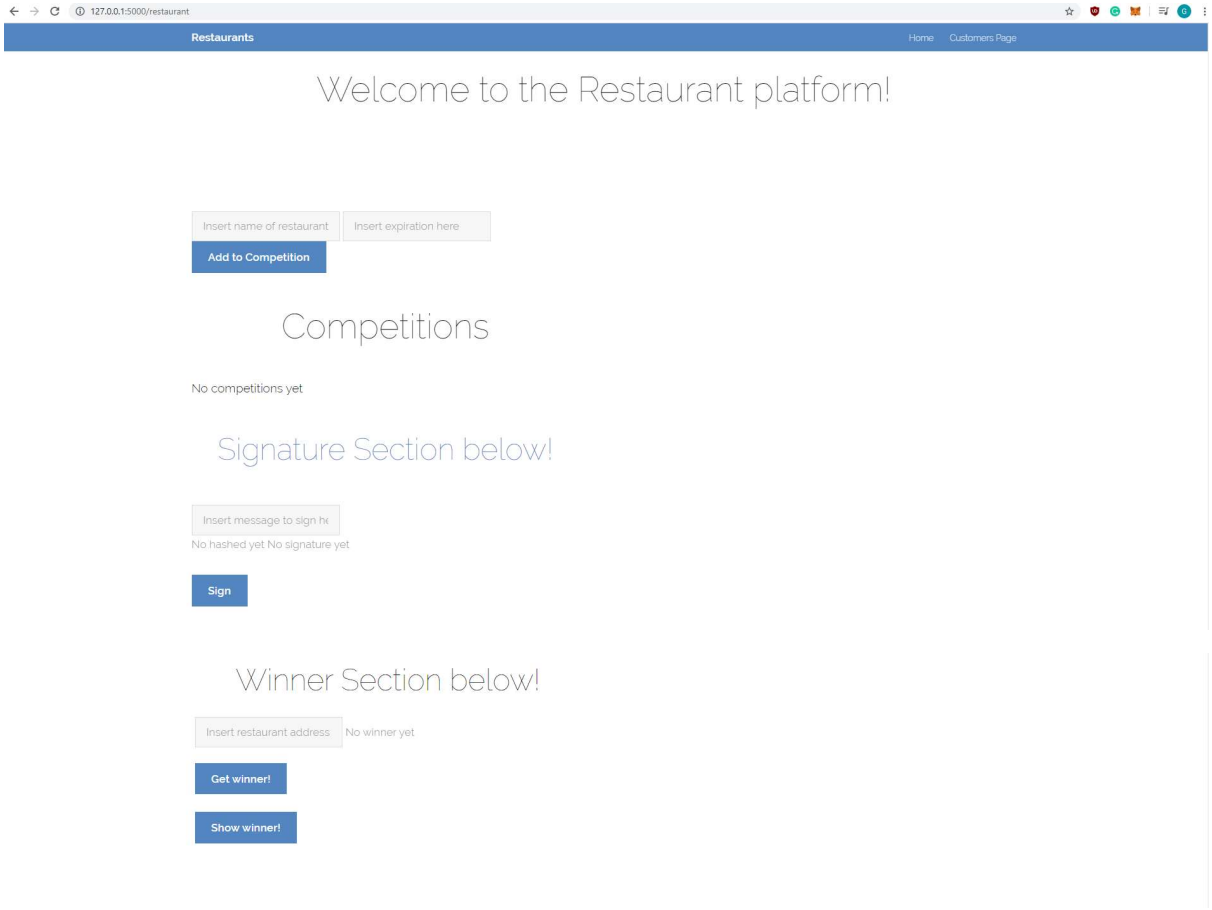
First, the owner compiles the solidity contract and using the transact method calling the constructor, creates a contract object. Then it sends it over as a parameter in the 3 render_template function that we use: the intro, the customer and the restaurant. At this point, we have 3 HTML webpages that are created, and everyone can access them. First of all, either a Restaurant or a customer will load the intro page, and after a 2 second Welcome

Screen that will be shown, he/she will be asked whether he/she is a restaurant or a customer:



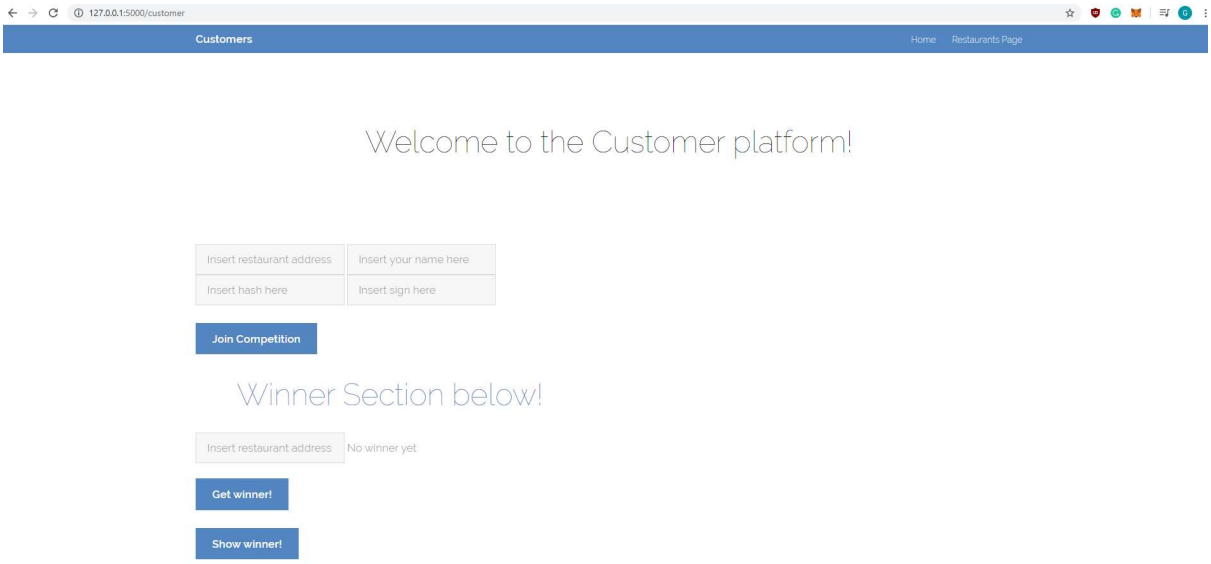
Each of the 2 buttons redirects the user to the corresponding webpage.

Below is the initial restaurant page, before any restaurant creates any competition:



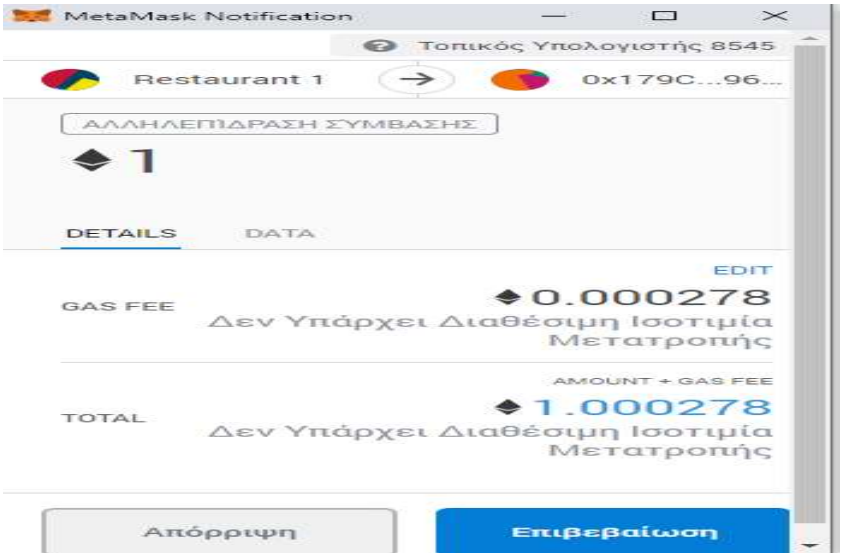
As it can be seen, below the competitions section, it says “No competitions yet”.

And below, this is the initial customer webpage:



Then let us say that a Restaurant wants to create a competition. First, it logs in with Metamask, inserts name of restaurant and expiration in seconds, and clicks Add to competition. For example, a restaurant with name: “Blockchain is Fun” wants to create a competition and sets up the expiration to be 10 minutes, which corresponds to 600 seconds. This is what it must do:

After clicking “Add to Competition”, we get prompted with the following from Metamask:



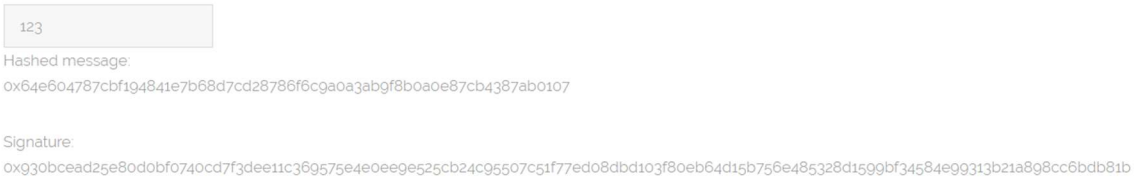
Note that the Restaurant 1 pays 1 ether to the contract (which will later be given to the winner), and the rest are paid as Gas fee. After clicking confirm, the competition is created, and we update the restaurant platform by including it:

Note that the "No competitions yet" got replaced with the above.

Then, let us say another Restaurant with the name: "My Dapp Rocks!" wants to join the competition, with again the expiration being 10 minutes. Then from the account of that restaurant in Metamask, we do the same procedure and the updated competition section looks like below:

Then, customer 1 named George visits Restaurant 1, Blockchain is Fun, before expiration. After finishing his food, in the receipt, the restaurant will hash the receipt ID and sign it with its secret key. He can do that via the web interface, by providing the receipt ID in the field, i.e. if receipt ID is 123:

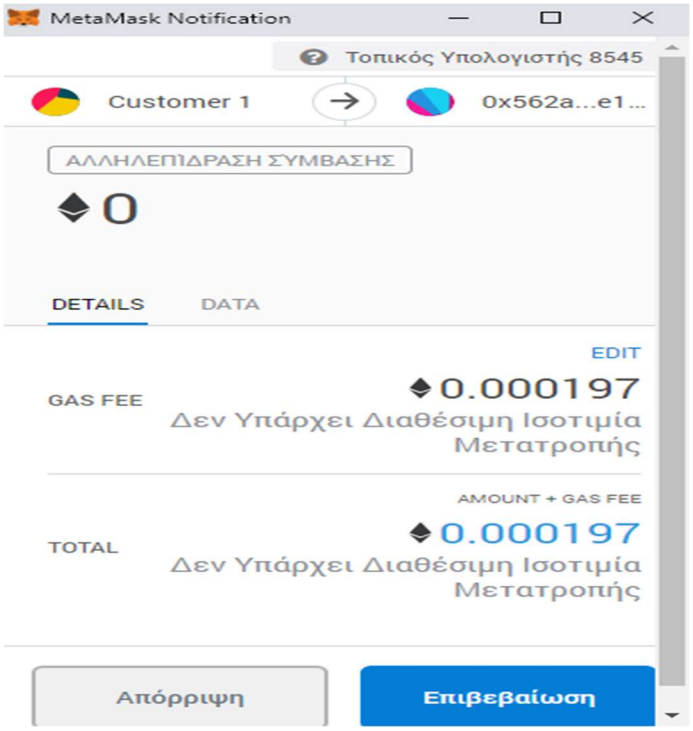
Then, it clicks sign, and we get the following:



It will then give the receipt to George that includes the restaurant address, the message hash and the signature.

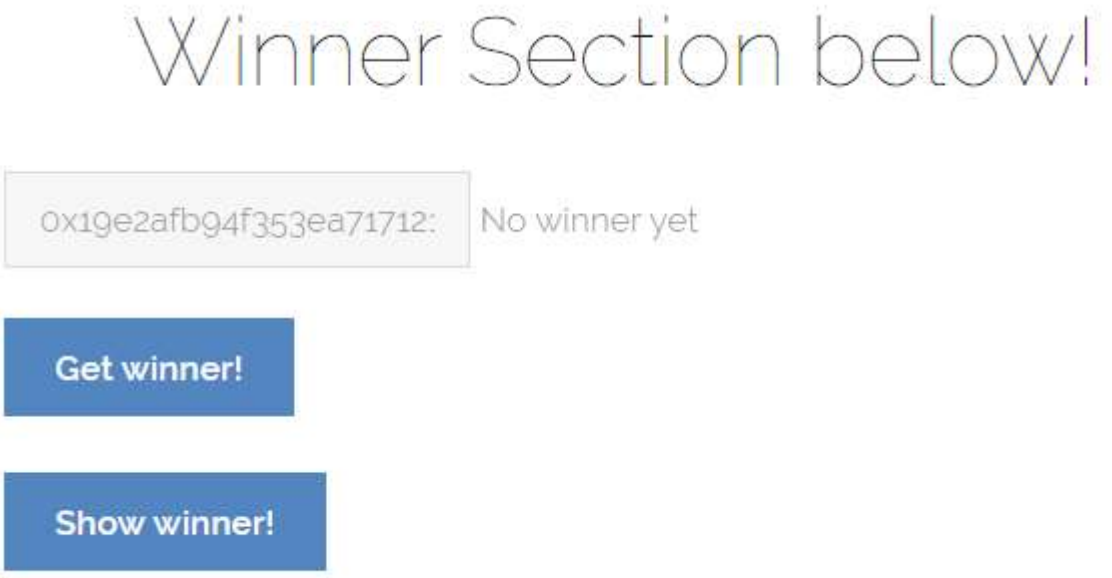
Then, George has an account in Metamask, loads the intro page, chooses he is a customer and wants to join the competition (we assume that 10 minutes has not passed yet when he will press Join Competition since we had used only 10 minutes for the expiration, he was very fast!). George inserts the restaurant address, his name, message hash and signature (as below) and presses Join competition:

After we press join, we get the following:

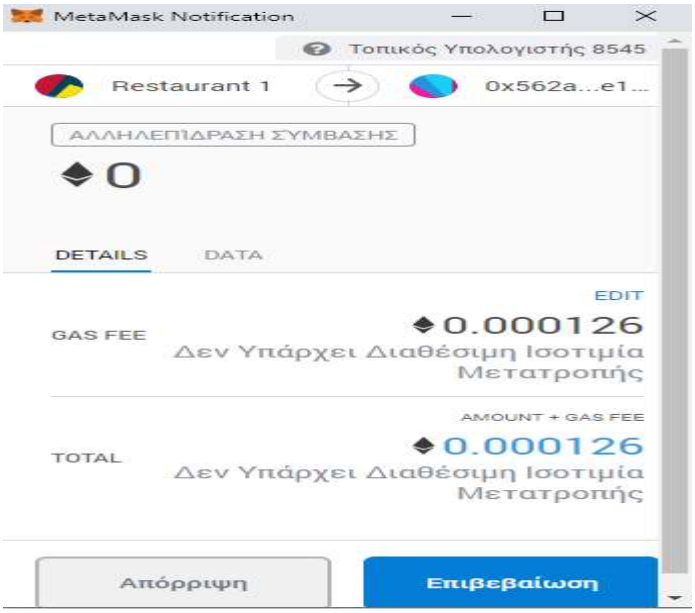


So, the customer pays 0.000197 Ether as gas fee and joins the competition. Many more users that have visited the restaurant after the creation of the competition and before the expiration, will get from the receipt the restaurant address, the hashed message and the signature from the restaurant and with those they will be able to join the competition.

The restaurant, before the expiration of the competition, inserts its address in the winner section:



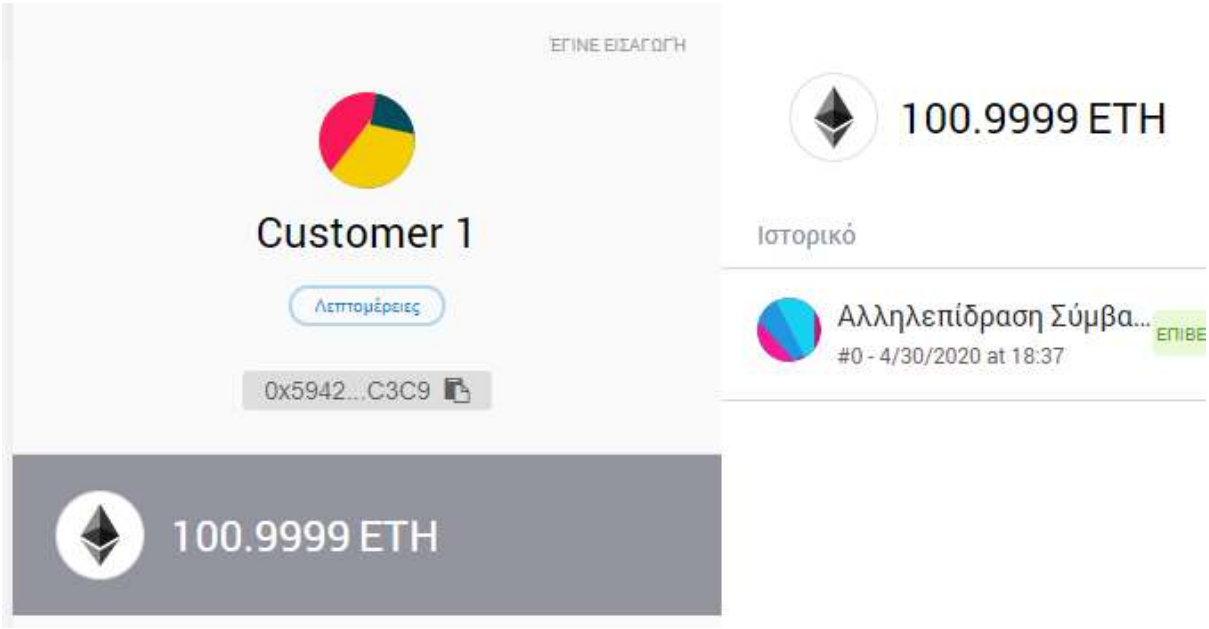
This is because on expiration, it will automatically call Get Winner with parameter the address specified in the above box. So, when it is called, we click confirm on the Metamask notification below:



Note that in case that either the restaurant did not include its address on the specified field when the function was automatically called, or rejects the transaction, it is not the end of the world. Either the same restaurant can press Get winner later, or any other restaurant or

customer can do that – It does not matter who does it, as long as the requirements pass, the outcome will me exactly the same. However, a “correct” restaurant would do it correctly in order to pay the gas fee.

At this point, the random winner receives the Ether from the contract automatically. In our case, we have only joined the competition with Customer 1, George, so he will certainly win:

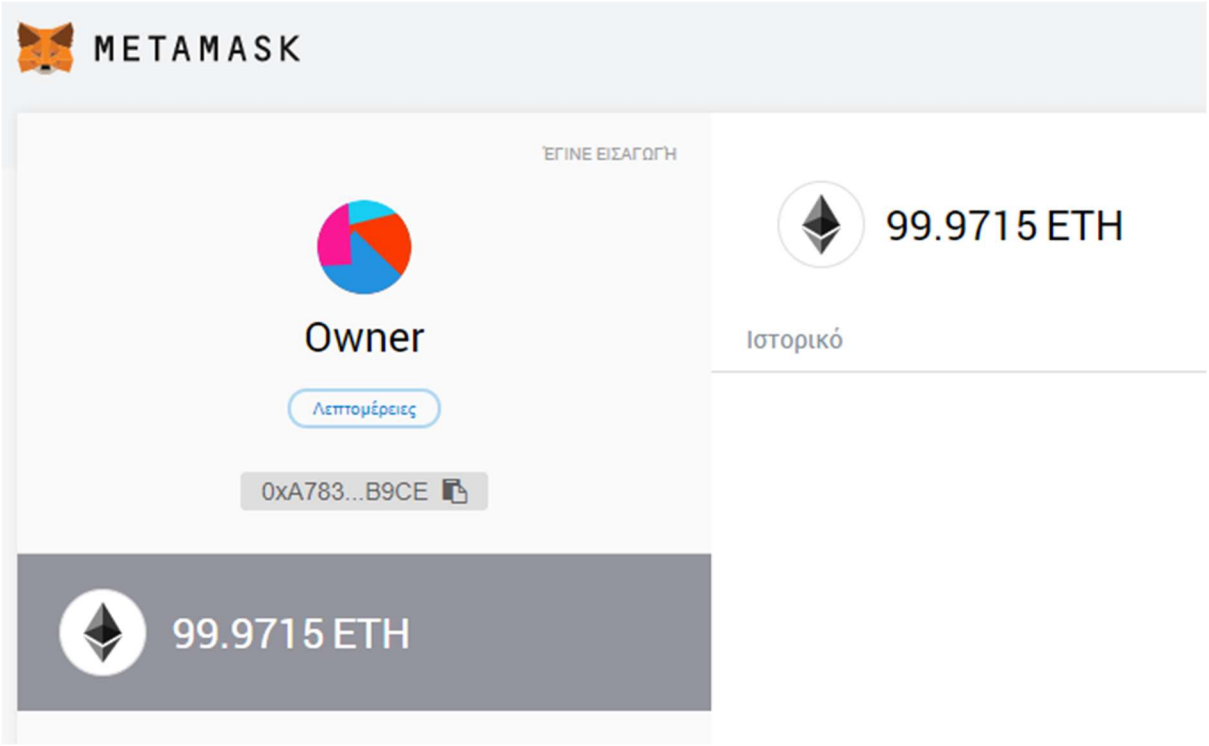


Even though our webpage does not yet show the winner, there is a view function in the solidity contract, and a button “Show winner” that triggers it in both the restaurant and the customer webpages. Since it is a view function, no one calling this pays any ether. The solidity contract is storing the winner of each competition once we call get winner. So, if we press “Show winner” before calling get winner, we will get the zero address. Otherwise, we see who won for that specific restaurant address provided in the box:

We can see that the winner's address is the same as in the screenshot above. The outcome is the same when it is called from either the Restaurant or the Customer platform.

ANALYSIS:

For the owner compiling and calling the transact method on the solidity contract:



From 100, it went down to 99.9715, so it has cost around 0.0285 Ether. However, compiling the same contract in solidity costs around $1x10^{-11}$ Ether in Remix. This is because in Remix, it does not multiply by the gas price. When we multiply with that, we approximately get the same value. For the compilation, what uses gas are the 3 global mappings – the restaurants, the customers and the winners – as well as the address of the owner and the storage of his/her address once the constructor is called. Just by writing the format of the two struct, as well as the event does not use any gas.

Note that in every explanation of what costs gas below, the storage typically uses a lot more than executing simple operations, and in my contract most operations(except the keccak256) are simple. Moreover, I have tested gas usage in all the following 3: Metamask+web3js, in Remix and in web3py(with the estimageGas method in web3py). After the last two were multiplied by the gas price as used in Metamask, which was 0.000000001 Ether, the results were all approximately the same.

The function add_a_restaurant_to_competition, when called from Metamask, and following a screenshot from some pages above, it costs 0.000278. In this function, some gas is used in the addition of an extra struct inside the storage of the mapping restaurants. Moreover, some gas is used in inserting the values to the struct Restaurant – Note that even though the Restaurant struct consists of a list of Customer structs, it is still empty so we do not use any gas for that.

The function add_customer_to_competition, when called from Metamask, costs 0.000197 Ether for gas. It is worth mentioning that every require cost some amount of gas from my testing – exceedingly small – but still uses. Not only because it needs to execute stuff, but just by inserting require(true) costs some gas. Also, this amount of usage is combined with the amount of usage needed to call the function recover, to verify the signature, as we have it in a requirement statement. In this function we create a Customer struct, we update values, we insert the customer into the restaurants struct and we update the length of list with the customers. All these assignments and storages cost gas. Note that we are for looping over all the previous customers that have already joined the competition to avoid any replay attacks, so every time a new user is added, the gas usage is increased.

The function get_winner, when called from Metamask, costs 0.000126 Ether for gas fee. The requirements cost very small amount of gas, the most amount in this function is used to perform the keccak256 of the current time and block timestamp and do a modulo with length in order to get the random number. Note that emitting the event seemed to be a bit costly as well – It used 0.000003 Ether – Testing my implementation just without emitting costed for this function only 0.000123! Then it also costs in order to transfer the money from the contract to the winner's wallet.

The function show_winner is a view function and does not cost any gas fee at all.

Recover by itself is a relatively cheap function costing 0.000029841 Ether. This is because it uses direct assembly. Note that the restaurant is signing within JavaScript and not solidity, I only verify the signature in the solidity code. Also, I could not calculate this with web3js+Metamask, since it is only called internally inside the solidity contract, so I have calculated in using web3py (the way I did it is commented out as being the last test in the Test data – for the Test Data, see next page).

Security:

The only time where there could be a way possible for a reentrancy attack to happen is when transferring Ether from the contract to the wallet of the winner. However, that is not really possible. Not only I use transfer instead of call that limits the amount of gas that can be used outside of our contract, but also all the checks happen first and transferring the Ether is the last operation that is performed. Moreover, I am not using any assert on the transferring of the money, since if for any reason the receiver of the money decides to use in his/her fallback function: throw; and will not accept the Ether, the contract won't fail. It will just continue to run and still have the money inside the contract.

Also, the way I have performed the randomness in my contract is almost random – not completely random. The block.timestamp is a bit in the control of the miner, and since I wanted to increase the security a bit, I have included in the calculation of the keccak the exact current time as well, since maybe there could be some time passed from when called until it reaches at that point that we cannot predict exactly. Still, this is not completely at random. As I read in the lecture notes, in order to implement complete randomness, I could do the following:

_seed = uint64(sha3(sha3(block.blockhash(block.number), _seed),now)), and then do a modulo on how many customers have joined the competition.

Furthermore, there is a for loop over all customers in order to make sure the customer has not already joined the competition and avoid any replay attacks. This potentially can run into out of gas exception if the customers are a lot. However, since it is important to check against replay attacks, the solution is not to avoid checking. A possible solution could be to test until a maximum certain amount that there is no out of gas exception and limit the amount of possible participants to that maximum value.

What has been done for improvement up to now, and what can still be done:

In my solidity code, I was using either bytes for the signature or bytes32 for the hash, and not an array of bytes since it wastes 31 bytes of space for each element due to padding rules. Moreover, I am using require instead of assert that uses a lot less gas. Furthermore, the fact that I am calling recover to verify the signature internally and not with a different call externally, saves a lot of gas since it just passes a memory reference to the internally called function.

An easy improvement would be for the customer not requiring to insert his name – it is just a useless extra information, that maybe could be better not to have it at all not only because of waste of gas, but also to enhance the anonymity provided from blockchain.

Moreover, instead of storing the name of the restaurant as a string, it could be stored as bytes32, since it wastes a lot less gas.

Furthermore, instead of using uint256 for the length of how many users will join the competition, it would be better to use something more realistic that will not overflow for sure as well. The most logical value would be uint16, since uint8 will only give space to 256 customers.

Another improvement would be not to store the address of the owner since my current Dapp is not using the owner anywhere except from deploying the smart contract.

Finally, it would be important after the expiration of the competition, to delete the entry of the Restaurant. This would not only save a great amount of gas, since storage and especially global storage costs a lot, but also it would give the restaurant the opportunity to create another competition after the expiration of the current one. By deleting, we will be deleting the storage of all the customers as well that have joined the competition, saving even more gas usage.

Test Data:

There is a folder called “Test” and inside there can be found two python files: “contract_test.sol” and “server_test.sol”. I have written my test data in web3py. Before someone can run the tests, the same preparation needs to be done as before running the web app:

For the owner, each restaurant's and each customer's account, I have first opened Ganache. From Ganache, I copy the private key and load it inside Metamask. Then, after inserting how many accounts I wanted, I renamed them and then used the option in Metamask: localhost 8545. It gives me 100 test Ether for every account.

For the tests, 5 accounts are needed: Owner, Restaurant 1, Restaurant 2, Customer 1, Customer 2.

Moreover, the signature from the restaurant was not done in the solidity contract, it was done in web3js. Only the verification was done in solidity. So, in order to get the hashes and the signatures, after inserting the accounts from Ganache to Metamask, we need to get for the tests one signature from Restaurant 1 and one signature from Restaurant 2.

To do that, we navigate to the folder where the main server is with the webapp, and we execute "python server.py". In Metamask, we swap to account of Restaurant 1. We go to the Restaurants platform, and in the signature section we insert: 123. We click sign. We click confirm on the Metamask notification. We copy the signature and we replace it in the "server_test.sol" file in line 8. Then, we swap to Restaurant 2's account, we do the same and we replace that signature in line 9. After doing that, we are good to go.

We navigate inside the "Test" folder and we run "server_test.sol". All the tests that I have implemented are running, and every time a test succeeds, it outputs a summary of what the test was and what happened. The tests that were supposed to fail (i.e. doing a replay attack), I included them in "try" "except" sections, so that when it fails, it prints that it failed and why.

This is an example of me running the tests:

```
(base) C:\Users\geoio\Desktop\SUTD courses\Blockchain Technology\app\RestaurantDapp\Test>python server_test.py
Added one competition for two restaurants successfully
Failed to add two competitions from the same restaurant
Added two customers to same competition
Failed to add customer with address of a restaurant that already has competition ongoing
Failed to add customer after expiration
Failed to add customer twice in the same competition
Failed to add customer with signature from someone else
The winner was the only participant in the competition
The winner was one of the two participants in the competition
Failed to call find a winner before expiration has passed
Failed to try to get a winner after a winner was already found
All tests pass!
```