CSC2125 Solidity Assignment

University Of Toronto, Winter 2024

January 31, 2024

1 Introduction

Most likely, you have heard about NFTs (non-fungible tokens). However, media coverage of this technology usually fails to describe their REAL use cases, focusing only on how speculators try to make money by selling images that can be copy-pasted using one click of a mouse.

However, if you think about it, NFTs can be applied to some areas of real life, such as tickets. One ticket represents a unique event that happens only once (e.g. CSC2125 lecture at 1pm on Dec 29th) and a unique place in the audience (seat number 1 in the front row), meaning that these tickets are not fungible.

We are asking you to code a ticket marketplace using smart contracts so that you can get familiar with Solidity and the tooling which helps develop Solidity smart contracts. Completing this assignment will give you a quick overview of the language constructs and show you how you would approach coding your smart contracts.

2 Assignment in more detail

2.1 High-level picture

Let's define the operations that our ticket marketplace should be able to perform. First of all, we should be able to buy tickets (duh). We should be able to do this using two currencies: native Ethereum currency Ether and ERC-20 tokens. ERC stands for *Ethereum Request for Comments*, which are documents that act as standards in the Ethereum community (just like RFC standards in networking). You can think of an ERC-20 token as a custom coin that you create on blockchain where you can program your own rules about how the coin works (real example from real life).

Prices for 1 ticket are set separately for each of the events, and each ticket should be bought for the same price. However, the price in Ether might be different from the price in ERC-20 tokens (e.g., 1 ticket for an event might cost 1 Ether, but in ERC-20 tokens it would cost you 1337 tokens).

The ticket marketplace should encode tickets as NFTs. We are specifically interested in the ERC1155 standard of NFT implementation since they allow us to encode each ticket as a pair $(event, seatNumber)$.

Quick overview of how we want to apply the ERC1155 standard: imagine that each ticket has a unique ID that uses 256 bits. We will encode the event ID inside the first 128 bits (bits 0-127 starting from left to right), and we will encode the seat number in the last 128 bits (bits 128-255). For example, to encode the ticket for an event with ID 3 and seat number 4 we would do the following calculations to assign it an ID:

$$(3 << 128) + 4$$

.. which is a pretty large number (note the parenthesis, bit shifts are done after the addition).

Also, we would like to have an option to limit the number of tickets that can be sold in total for each of the events. This could be useful when the organizers see that the event is popular and they can try to fit in the same venue to get more revenue. You would have to account for the fact that only the administrator of the smart contract should be able to update this information since Ethereum is full of people trying to hack your smart contract.

We would also want to be able to update the address of the ERC-20 token that we use to buy tickets, and the prices themselves.

Each function in your smart contract (besides the constructor function of the smart contract and functions that correspond to buying tickets) should be accessible only by the contract creator. Also, each function besides the constructor must emit an event after it performs a desired action.

Note that you don't have to implement ERC-20 and ERC-1155 standards from scratch; OpenZeppelin provides an audited version of these standards, which you can then customize to your needs. Here is a quick example of how you would create your own NFT contract (note that this contract should be separate from the marketplace contract).

2.2 Some details about starting code

You need to code a smart contract using Solidity programming language. We will be giving you the skeleton of the code (interfaces of the smart contract functions) and the full test suite that uses Hardhat development environment and Typescript language (no way around this language if you are doing something in Web3, sorry). If your code passes all test cases, you can consider the assignment to be completed.

We would require you NOT to modify the test cases provided (unless you find a bug in tests; in this case, please report this to Piazza or TA's email and we will take a look). Any modification to the code of the tests will be considered as cheating and your mark for this assignment will be reduced to 0.

We will also ask you to refrain from hard-coding the test cases in the smart contract; for example, your solution should be generalizable for an arbitrary address that (a) uses the system; (b) is the owner of the smart contract (i.e. if you perform access control only on 1 address that is being tested by the test suite, your mark will be reduced or become 0). The same goes for the amount of tickets that can be bought, prices in the smart contract, and other things that you will find in the test cases.

2.3 Setting up the project

First, install Node.js: link. To set up the project, unzip the archive CSC2125_HW_TicketMarketplace.zip and run this inside the folder you have just unarchived:

```
npm install
```

To run the test suite:

```
npx hardhat test
```

In case you need to update the generated types inside the Typescript code (in case you want to mess around with a copy of the file with a test suite while coding your solution):

```
npx hardhat compile
```

This step is implicitly done every time you update your Solidity code and run the test command.

In case you would like to debug your Solidity code, Hardhat provides you with an ability to use console.log statements inside your Solidity code.

2.4 What to submit

Please submit the whole project as a link to the GitHub repo (please don't delete the .gitignore file). Your solution should be testable with the test command specified in section 2.3.

2.5 Small hints

Some advice here.

Hint 1: **read tests carefully** and try to understand what this test is doing.

Hint 2: imports in the Solidity files should send you in the right direction :)

Hint 3: Use Google in case you hit a concept you don't know or ask on Piazza! Don't get stuck for long periods of time.