Imperial College
London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Principle of Distributed Ledgers

Author:
Tianquan Zhang (CID: 01755554)
Email: tz219@ic.ac.uk

Date: February 18, 2020

1 Introduction

CryptoMons are collectable and breedable digital Pokemon cards. It is based on the famous game CryptoKitties. When two CryptoMons breed, their statistics (e.g. HP, Attack) are determined by the statistics of it's species and parents.

At launch, the player picks one of the three starter Pokemon: Charmander, Squirtle or Bulbasaur (as shown in Figure 1). The starter Pokemon is essentially free, where user only pay for the gas required for the transaction.

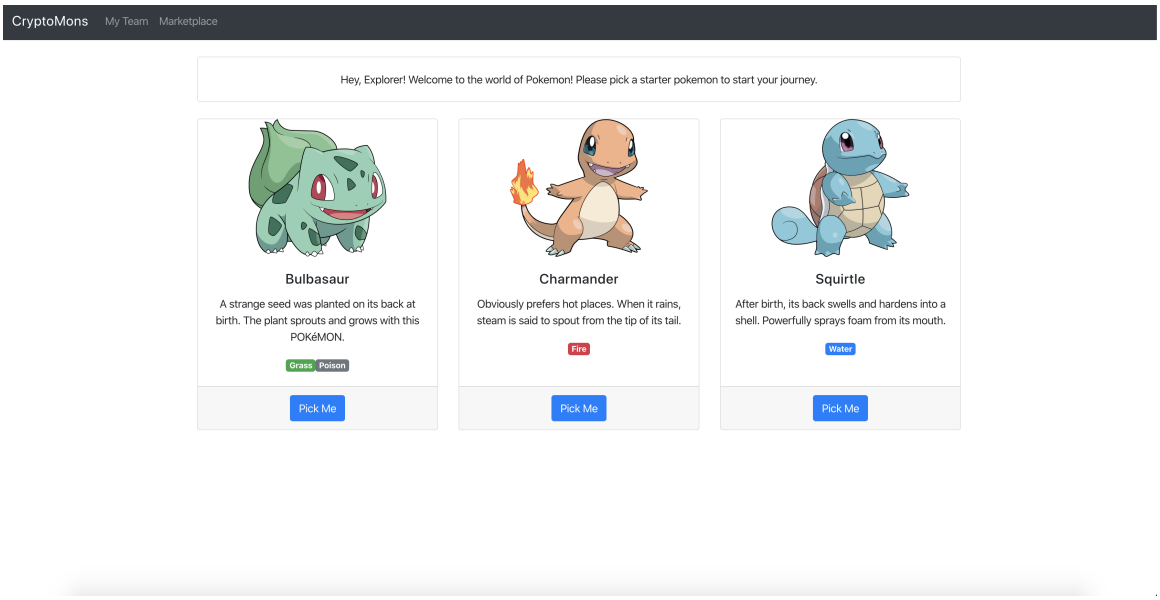


Figure 1: Starter Pokemons

There are two three activities in CryptoMons: Breeding, Battling and Trading. These are the main ways in which the user can obtain more Pokemon. As we can see from Figure 2, player are can also buy Pokemon from starter packs. These are offered in the Pokemon shop. As Pokemon battles other Pokemon, they can level up and trade for a higher price in the marketplace.

Breeding in CryptoMons is designed to mirror breeding in the real Pokemon game. Breeding requires a male and female Pokemon to participate. Gender is assigned to the Pokemon during creation and cannot be changed after. Once they start breeding, the mother Pokemon becomes pregnant and cannot battle during this time. After a certain amount of time (measured in blocks) have past, the Pokemon has the option to give birth to a new Pokemon that is the same as the mother. However, the new born Pokemon will not be the highest evolutionary form.

The marketplace is one of main game elements that allows players to buy Pokemon. It shows all Pokemon currently on the market. No approval of the buyer is needed from the seller once the Pokemon is on the market. As a result, anyone can buy the CryptoMon once it is available and the quickest to get their transaction through becomes the new owner.

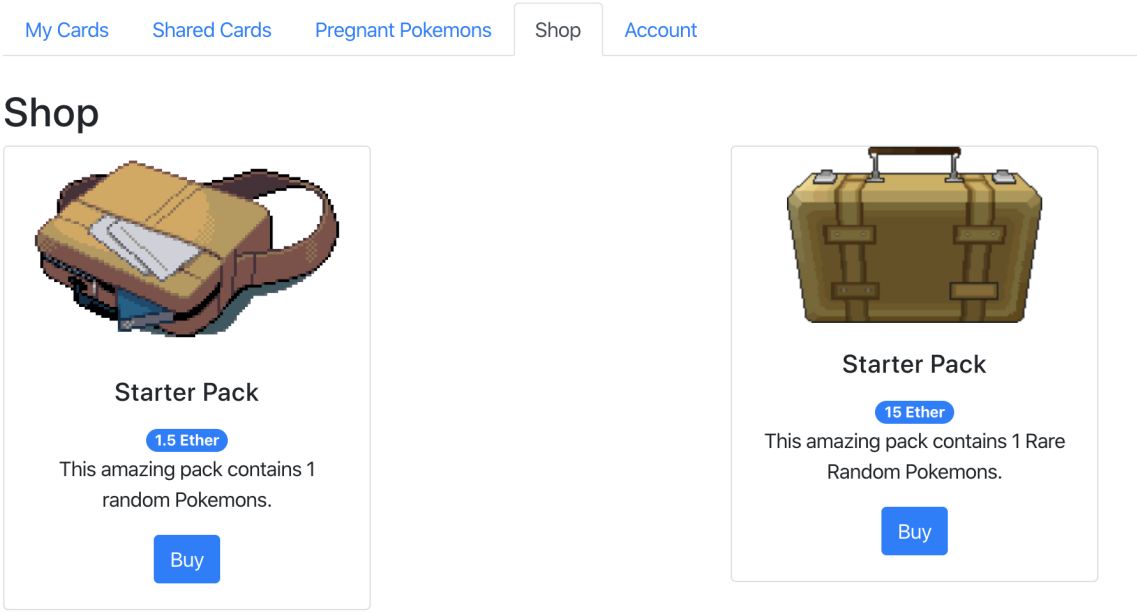


Figure 2: Shop

Each player have their personal space on the platform where they can view the CryptoMons they own (as shown in Figure 3). This includes statistics like health point, attack, defence, gender of the CryptoMons. In addition, they can decide whether they want to sell/share them or not. Four options are available:

- **Sell**: The Cryptomon becomes available on the marketplace and the ownership will be transferred once another player buys it.
- **Stop Sell**: The Cryptomon is taken off the marketplace and becomes unavailable to buy.
- **Share**: The Cryptomon is shared with another player. Hence, the other player can use the shared Pokemon to battle.
- **Stop Share**: The Pokemon stops being shared. Other player can no longer use it to battle.

In CryptoMons, Pokemon can battle other Pokemon. The first Pokemon to run out of health loses! When a Pokemon wins a battle, it levels up and its statistics increases by a small amount. Each time the Pokemon level up, there is also a small chance that it will evolve into a different species of Pokemon if it's not in its final evolutionary form yet. When a Pokemon evolve, it's statistics can increase significantly or none at all, depending on its current level. Besides, its can sometimes change types.

To share the fun, Pokemon can be shared with other players, using their address. The address can be obtained from the account page. Using the shared Pokemon, other players can use it to battle. In other word, other Player can help you level up and even evolve your Pokemon!

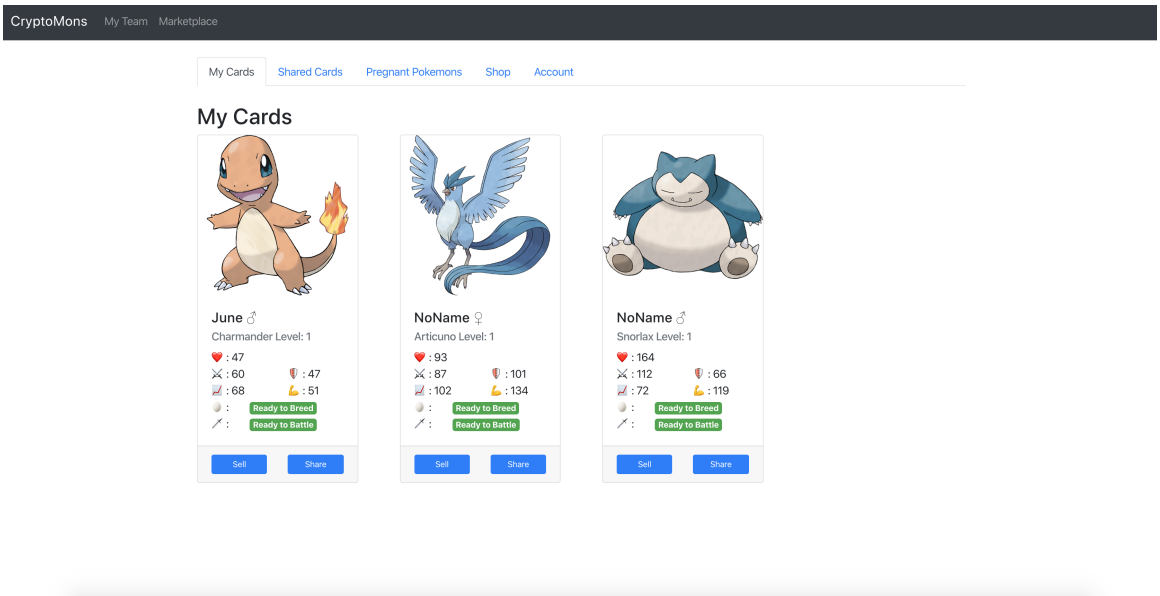


Figure 3: Home Page

2 Architecture and implementation

2.1 Solidity contracts

The code for CryptoMons is decoupled into a number of smaller contracts. Each extends the functionality of its parent with a major feature. Multiple-level inheritance keeps related code bundled together, and reduces the amount of code in each contract. The inheritance for CryptoMons is shown below:

```
contract Base is Ownable
contract PokemonCardFactory is Base
contract PokemonCardShare is PokemonCardFactory
contract PokemonCardBreeding is PokemonCardShare
contract PokemonCardHelper is PokemonCardBreeding, ReentrancyGuard
contract PokemonCardAttack is PokemonCardHelper
contract PokemonCardOwnership is PokemonCardAttack, ERC721
contract PokemonCardMarketplace is PokemonCardOwnership
```

The `Base` contract extends the `Ownable` contracts written by OpenZeppelin, which provides numerous reusable functions and modifiers to create secure smart contracts e.g. `transferOwnership` transfers the control of the contract to a new owner.

The `Base` contract manages various information different species of Pokemon. It includes Pokemon number, types and base statistics of each species. The contract also defines whether a Pokemon can evolve into a different species of Pokemon.

The `PokemonCardFactory` contract contains the most fundamental code. It consists

the main data storage, constants, structures and functions that manipulates these data items. It defines Pokemon as a struct:

```
struct Pokemon {
        string nickname;
        string type1;
        uint32 level;
        uint32 breedingReadyTime;
        bool gender;
        uint dna;
        uint fatherId;
        uint motherId;
        uint breedingWithId;
        uint pokemonNumber;
        BaseStats baseStats;
        ...
}
```

Breaking each into:

- **nickname**: The player defined name of the Pokemon.
- **type1**: The type of the Pokemon e.g. Grass.
- **level**: The current level of the Pokemon.
- **gender**: True represents male and False represents female.
- **breedingWithId**: The field is set to the ID of the father when the Pokemon becomes pregnant. Otherwise, it's set to zero.
- **fatherId & motherId**: the ID of the Pokemon's father and mother respectively.
- **dna**: the main piece of code that determines the statistics of the Pokemon.
- **breedingReadyTime**: the current pregnant duration before it can give birth to a new Pokemon.

The contract declares an array of Pokemon structs `Pokemon[] pokemons`. The array contains all Pokemon ever created by the contract. Whenever, a new Pokemon is created, it's pushed to the back of the array and it's index becomes it's ID. A mapping also exists which maps from the ID of the Pokemon to the address of it's owner. This keeps track who owns each Pokemon.

The `PokemonCardShare` contract contains the functionality to share the Pokemon to another player for battle. There are two main functions `shareCard` and `unshareCard`. By default, all Pokemons are shared with address 0x0. The zero-account is just a

special case used to indicate that the address does not exists yet. When an owner wish to share their pokemon with another player. The value is changed. When an owner no longer wants to share the card, the value is reverted back to address 0x0.

The `PokemonCardBreeding` contract includes the core logic to breed two Pokemons. To get a baby Pokemon, there are two separate steps:

1. Get a female Pokemon pregnant with a male Pokemon. The function takes the ID of the mother and father, and performs various checks i.e. to make sure the mother is not already pregnant or they are inter-breeding between siblings. Afterwards, the gestation period begins for the mother.

2. Call `giveBirth` function when the gestation period is done. Further check is done to make sure the CryptoMon is ready to give birth. The ownership is assigned to the mother's owner, and `_breedPokemon` function defined in `PokemonCardFactory` is called.

Moreover, the `PokemonCardBreeding` contract also contains the logic to abort the pregnancy. This is triggered when the pregnant Pokemon is attacked and loses.

The `PokemonCardHelper` contract provides the functionality to level up the Pokemon, and evolve the Pokemon. `evolve` is only triggered after the Pokemon leveled up. In addition, it includes functions to buy starter and rare packs from the shops, which both offers a random Pokemon to the player at a cost. Pokemon from rare packs have higher probability of being legendary. The ether gained are transferred to the owner of the contract to help with further developments. It inherits from `ReentrancyGuard`. This results in the availability of the `nonReentrant` modifier. The modifier ensures there is no nested reentrant calls to the function.

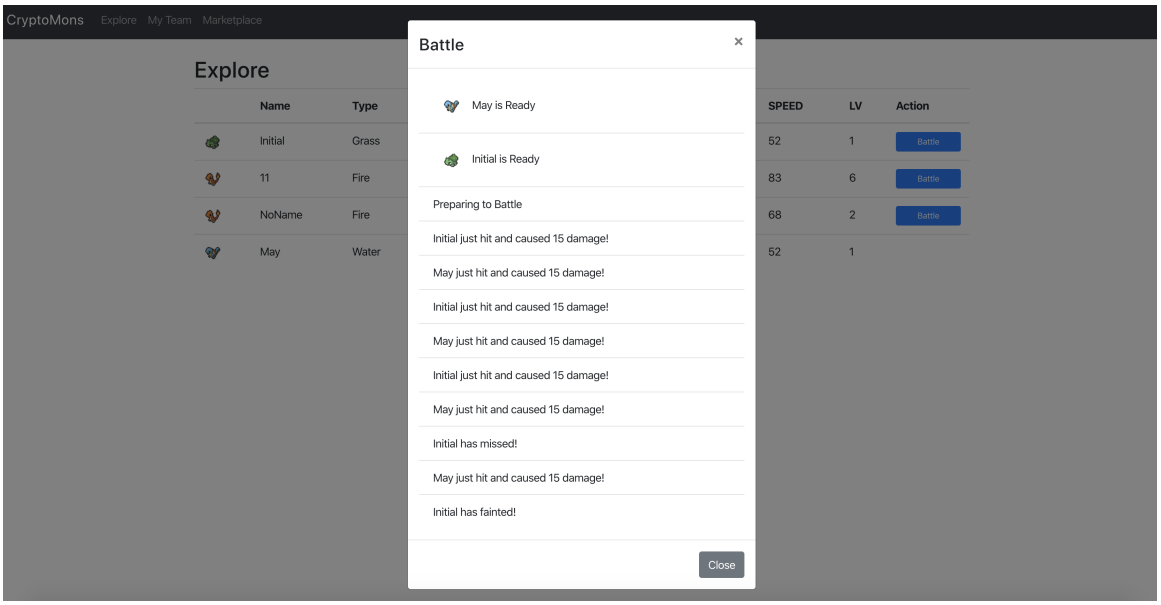
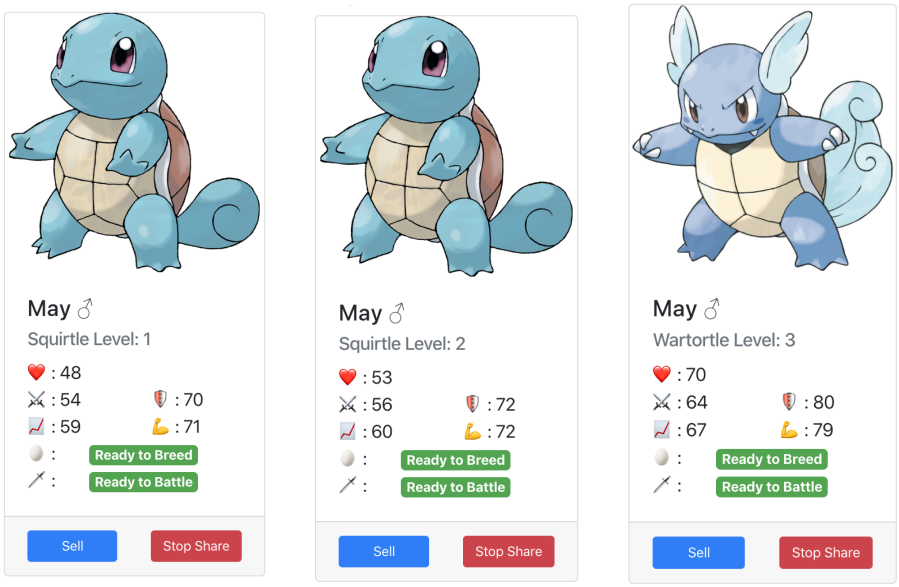


Figure 4: Battle Result

The `PokemonCardAttack` contract manages the battle logic between Pokemon. The battle result is shown to the player in the form of turns. As we can see from Figure 4, each turns consists the damage incurred and the Pokemon that attacked. The damage is not deterministic. Instead, a random number is obtained each turn. If it is larger than the speed of it's opponent, damage is incurred based on the Pokemon's statistics. Otherwise, the Pokemon will miss the attack. The winning Pokemon levels up. There is also a small probability that the Pokemon will evolve. This is illustrated in 5, the Squirtle in Figure 5(a) leveled up after winning the battle shown in 4. After a second successful attack, it leveled up and evolved (as shown in Figure 5(c)).

The `PokemonCardOwnership` contract provides the implementation of methods necessary for basic non-fungible token transactions. CryptoMons conforms to the specification of ERC-721 (Ethereum Request for Comment 721). This allows the CryptoMons to be treated as tokens which are not fungible. In other word, every CryptoMon is not created equal. They cannot be interchanged with each other, even though they have the same value.



(a) Level 1 (b) Level 2 (c) Level 3

Figure 5: Level Up

`PokemonCardMarketplace` is the contract address the app points to. It inherits all the data and methods from the parent contracts. There are three main public functions:

- **createItem**: The user creates a listing in the marketplace, which consists of the Pokemon that they wish to sell and their desired price.
- **buyItem**: Buy the Pokemon listed.
- **takeOffMarket**: The user no longer wish to sell the Pokemon. The listing is taken off the marketplace, and is no longer visible in the marketplace.

2.2 Web interface

The front end of the application is built using React and web3 library (Ethereum JavaScript API) which provides an API to interact with the blockchain. Metamask is also used during development. It takes care of web3 injections, signing transactions and connecting to the right node. By clicking on specific buttons or filling out forms, React sends the request to the blockchain using web3 and MetaMask. A response will be progagated back to the frontend, which will then render to the user.

Using web3, events (i.e. `NewPokemon(uint pokemonId, string name, uint dna)`) on the Ethereum blockchain can be queried and subscribed. Hence, specific actions can be triggered when certain criteria are met. In other words, it allows React to render an update to the platform in real time. User do not need to refresh the page, to view a change. For example, An listener can be created to listen to upcoming new Pokemon events. When the user buys a new Pokemon from the shop, the event will trigger a callback that adds the new Pokemon onto the user's personal page. No page refresh is needed.

3 Threat model

Integer overflow and underflow is an issue where a numeric value is produced that is outside of the range that can be represented by integer. As a result, we used `SafeMath` to perform arithmetic operations such as addition and subtractions.

Reentrancy attack is also a major issue. It is where an important state variable is altered after the contract invoke another function. In these situations, attacker can call back into the original contract and invoke the same function recursively. To avoid reentrancy, a simple method involves updating the state before invoking a function that is not owned by the contract. Another variation is Cross-function race conditions. It occurs when multiple functions share the same state. For example, in a contract where the balance is only reset after the transfer occurs. The attacker can call the function recursively even after withdrawal was made, to obtain more ether than they actually own. In our implementation, we reordered our code to make sure any external calls are made after the state change. In addition, we used OpenZeppelin's ReentrancyGuard implementation, which guards our withdrawal function with a mutex.

Block gas limit was also a large consideration when implementing the game. The Ethereum network only allow a certain computational load. The limit is conceptualised in the form of block gas limit. The transaction will fail if the limit is surpassed. As a result, some functions within our implementation were broken down into smaller operations. For instance, we provides several functions to retrieve different information about a Pokemon, instead of a large function which returns all meta information.

Since Ethereum is decentralised, it relies on computation power of distributed nodes, as compared to a centralised server to process the requests. Hence, nodes can synchronise time only to some degree. It can therefore be manipulated to circumvent any logic within our contract that relies on timestamps. Taking these into consideration, we mitigate these attacks by avoiding the use of timestamps within our contracts, especially for randomization seeds.

Another issue due to decentralisation is that random numbers are difficult to generate. Every algorithm for creating random numbers is pseudo-random. This means that someone with enough computation power can generate transactions and blocks which control the random number produced. Especially, timestamp and block hash can be influenced by miners. A viable method to produce random number is using a commit reveal scheme. The scheme consist of two steps:

- **Commit**: Generate a unique hash for your random number generated and distribute it to the public.
- **Reveal**: After all hashes are collected, an operation is performed on the hashes to produce a new random number. The public can verify that your submission and hash match.

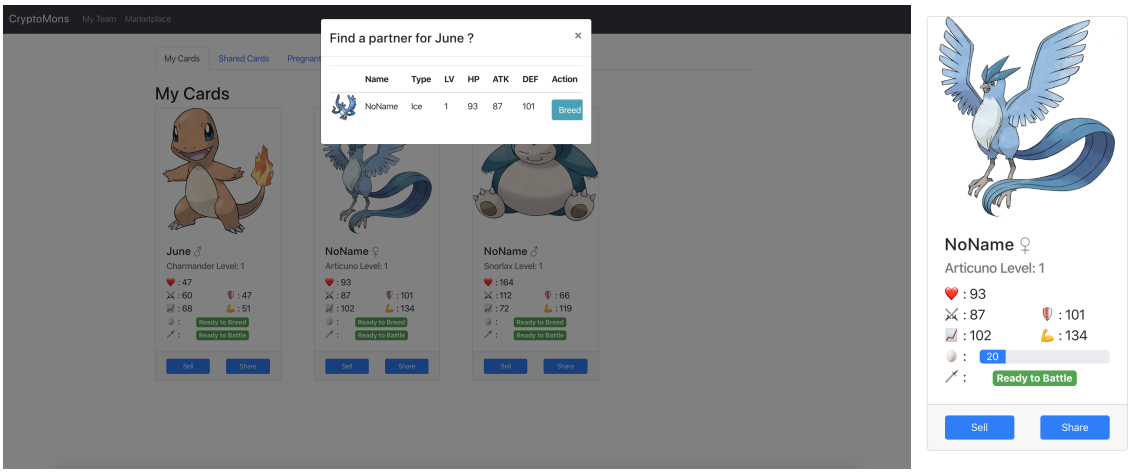
A problem with the scheme is the complexity. I think it is not necessary due to the gas cost. Instead, we kept a global variable called `nonce`, that increments each time a random number is generated globally. The variable is hashed with other variables and the modulus is taken. The reasoning behind the approach is that `nonce` can be influenced by other people, so it is harder to control or hack.

4 Final result: CryptoMons

In this section, we will walk through three main features of CryptoMons.

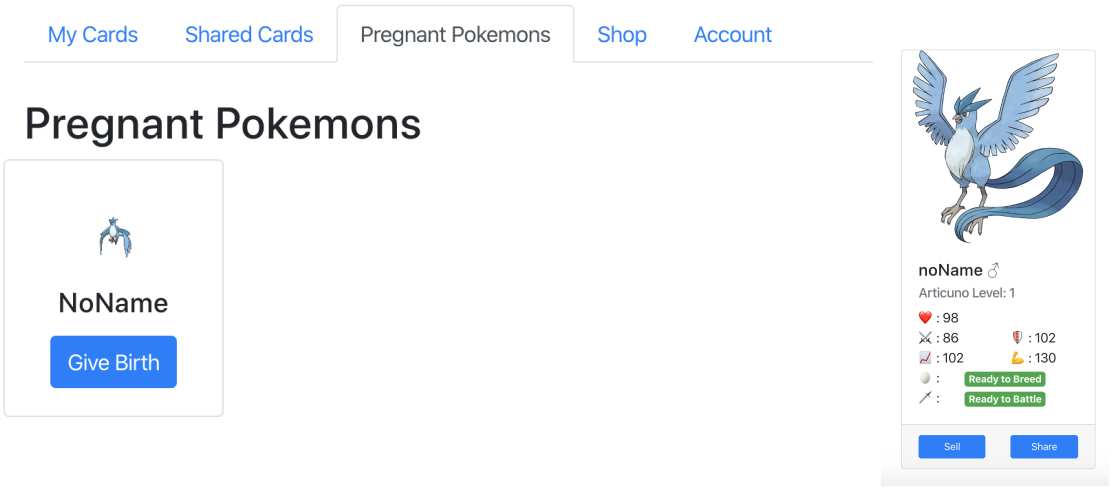
Each Pokemon in the home page is tagged with a button that indicates whether a Pokemon is ready to breed. If it is ready, a click on the button will spawn up a dialogue (as shown in Figure 6(a)). The user can then choose through a list of their Pokemon, which are opposite sex to the specified Pokemon. After the user submitted the request, the tag is changed asynchronously to prevent further matching. The tag is replaced by a progress bar, which indicates the time remaining until the Pokemon is ready to give birth. Under the tag ”Pregnant Pokemon”, the Pokemon will also be visible. After the required gestation period is over, a button to ”give birth” will become available (Figure 7(a)). Clicking on the Button, a new baby Pokemon is born (as we can see in Figure 7(b)).

We will now describe the marketplace, and how our contract is called. Before the Pokemon becomes visible in the market, a player must first list their Pokemon with a set price. This can be done by clicking on the sell button in the home page and



(a) (b)

Figure 6: Picking a partner



(a) (b)

Figure 7: Giving Birth

fill out the asked price. The form is shown Figure 8. Afterward, the Pokemon will become available on the marketplace, shown in Figure 9.

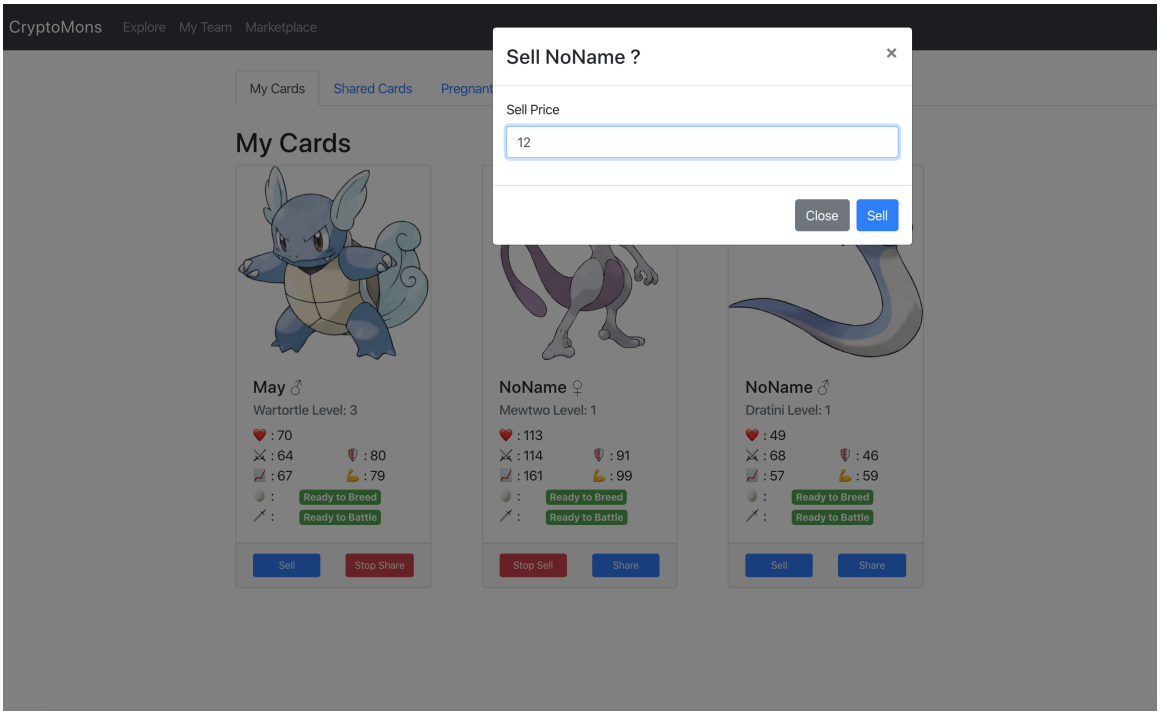


Figure 8: List the Pokemon

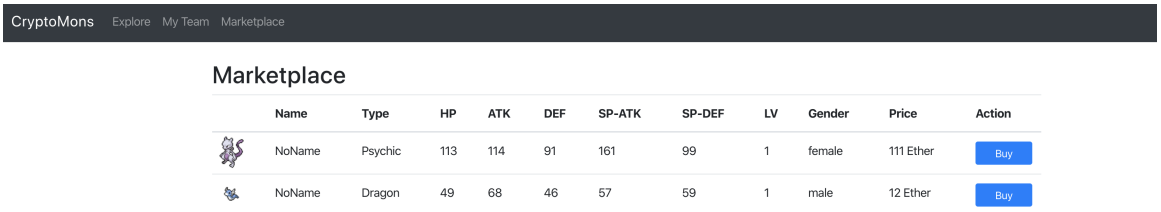


Figure 9: Marketplace

Another player can purchase it by clicking on the buy button. A callback is triggered, and a request is made to our contract using the web3 library. The code can be seen in Figure 10. Inside ethereum, a call is made on `_transfer` which transfers the ownership of the card to the buyer and the profit is transferred to the Pending Withdrawal balance of the seller.

In other words, the web3 library will make a call to our contract using Metamask, a bridge application, which is used to connect to the user account. A transaction is then made. Approval is necessary on Metamask. After inspecting the gas price, and amount, the transaction can be submitted (as seen in Figure).

```
buyPokemon = async(index) => {
  const itemId = this.state.marketplaceItems[index];
  const item = this.state.marketplaceItemsInfo[index];
  const { accounts, contract } = this.state;
  const amountToSend = this.state.web3.utils.toWei(item.price, "ether");
  const response = await contract.methods.buyItem(itemId).send(
    {from: accounts[0], value:amountToSend}
    ).then(function(res){
      console.log("buyPokemon " + res);
    });
  console.log("buyPokemon " + response);
}
```

Figure 10: Code to trigger transfer

(a) Metamask approval

(b) Transaction receipt

Figure 11: Transaction

Since, only the local server was used, instead of the Ropsten Test Net, we can not use Etherscan to visualise it's success. Instead, we access the receipt through the truffle console. From the transaction receipt in Figure 11(b), we can see that the status is true, indicating a Success. As a result, a `SoldPokemon` event was emitted and the newly purchased Pokemon becomes visible on our personal page (Figure 12). Under the account page of the seller shown in Figure 13, we can see that the profit is available for withdrawal from our pending withdrawal balance.

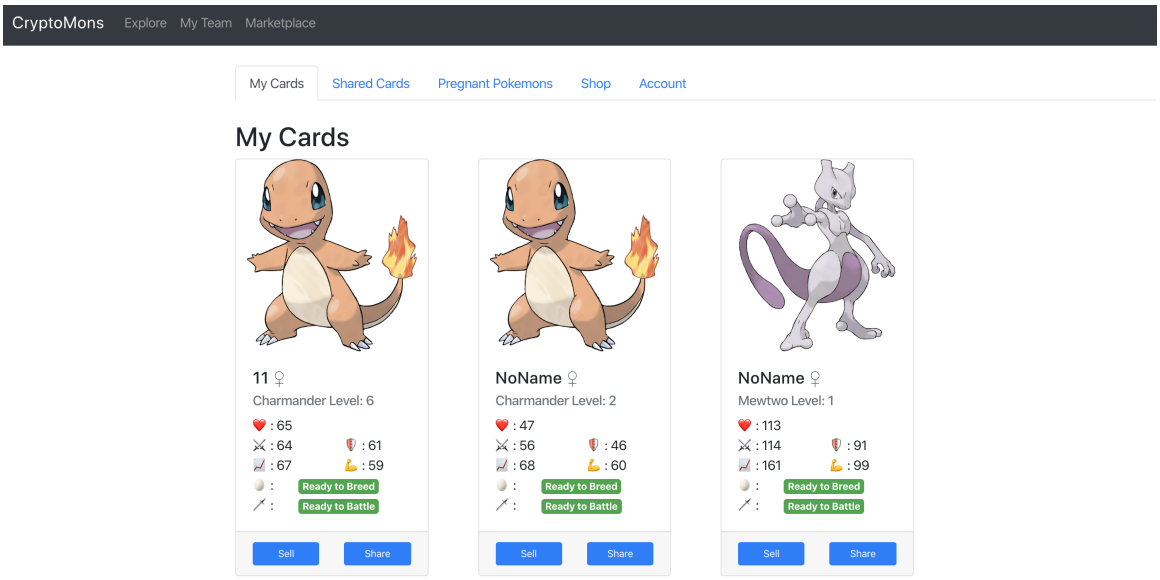


Figure 12: New Purchase Pokemon

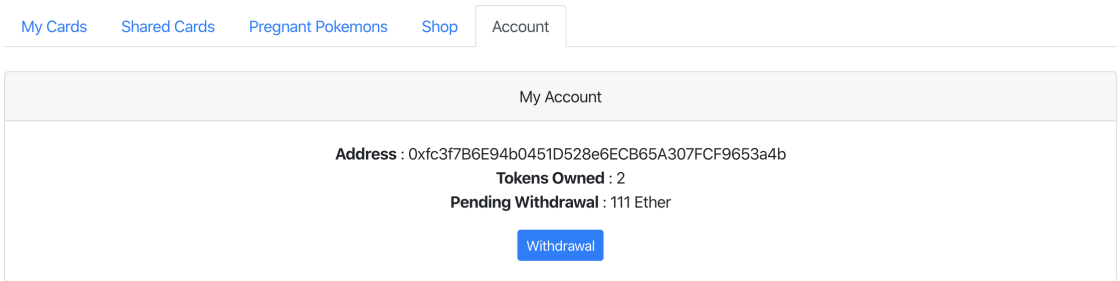


Figure 13: Account page