The Project Report for

“DApp: Pet Shop”

Submitted in partial fulfilment for a course

APS1050– Blockchain Technologies and Cryptocurrencies

Submitted by

Vijaykumar Maraviya (1006040320)

Guided by

Professor Sabatino Costanzo-Alvarez Instructor, UofT

Professor Loren Trigo-Ferre Instructor, UofT

Department of Mechanical and Industrial Engineering

University of Toronto

Executive Summary

In this project, I built Pet Shop, a distributed application (DApp) to buy and sell pets on the Ethereum blockchain. While the idea of the app is motivated by the classroom example of Pete's Pet Shop, the functionality is different; hence, the code is written from scratch. Pet Shop offers five major functionalities: filter and view pets up for sale, buy a pet, sell a pet, add a pet, and display account detail of a pet owner. To write, compile, and test the smart contract, the Truffle framework along with the Ganache test blockchain is used. The smart contract is written in Solidity and tested using JavaScript. Python web-development framework Flask is used to generate and serve HTML pages for the front-end. Flask talks to the blockchain using 'web3.py' to perform the back-end logic. The attached video with this report demonstrates the functionality of the application.

Contents

1 Introduction

As distributed ledger technology like Ethereum grows in maturity and becomes technically capable, the DApps (also called smart contracts) are becoming more useful and popular. Most of the current use of this technology today is observed in the financial service sector. The DApps provide services for lending and borrowing funds, trading assets and financial instruments, earning interest in the savings-like account, all of these without a centralized intermediary. With the rise of gambling Dapps and Non Fungible Tokens (NFTs), consumer adoption is also proliferating. Many developers are exploring the possibility of building decentralized e-commerce DApps to eliminate the traditional gate-keepers of today's online commerce.

Exploring along with the idea of building an e-commerce DApp, I present in this project report the Pet Shop, a proof-of-concept DApp to buy and sell pets on Ethereum (test) blockchain. Pet Shop offers anyone with an Ethereum account to buy and sell pets. In the following sections of this report, the app's features, smart contract code, and user-interface code, respectively, are explained in detail. Section 2 describes the features with screenshots of the user interface. Section 3 describes smart contract code written in Solidity and testing code written in JavaScript. Section 4 explains the working of the Flask app that serves as the front-end. Web3 python library (web3.py) acts as a glue between the Flask code and the smart contract deployed on the blockchain. The use of Web3 is explained in both Sections 3 and 4.

2 Features

Pet Shop offers the following five major features: 1) account detail of the user 2) filter and display pets up for sale 3) buying a pet 4) selling a pet 5) adding a pet. How a user can use these features is explained below.

2.1 Account Details

The image below shows the screenshot of the homepage, which users interact with when they visit the app.

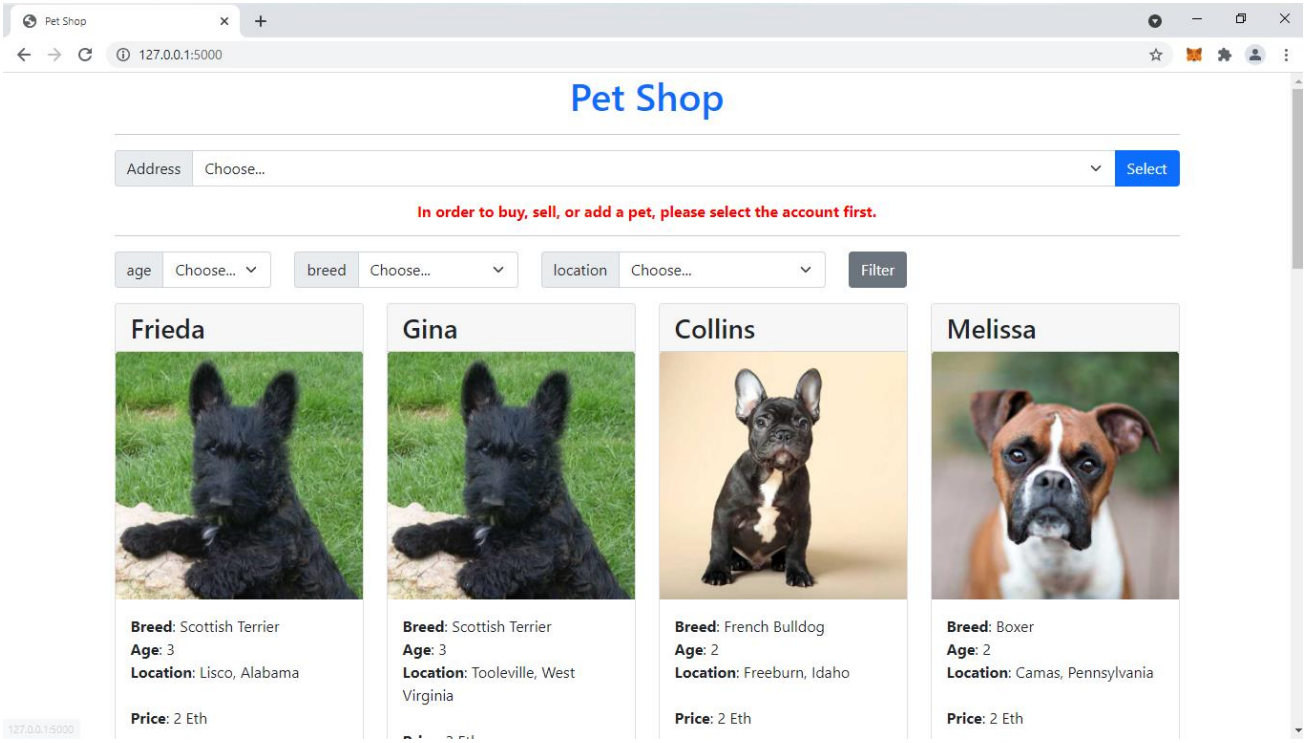


Figure 1 Homepage (screenshot)

The page provides the dropdown menu for users to select an account. It also displays the pets that are currently up for sale and the dropdown menus to filter them according to the value of attributes like age, location, and breed. However, a user cannot click the Buy button before selecting the account.

The image below shows the screenshot of the account page. In addition to listing the pets up for sale as before, it now shows the account information, including the selected address, balance, and pets owned by the account. It also shows the buttons for selling and adding a pet which redirects to appropriate pages when clicked, as explained later in this section. Also, note that the Buy button is clickable now.

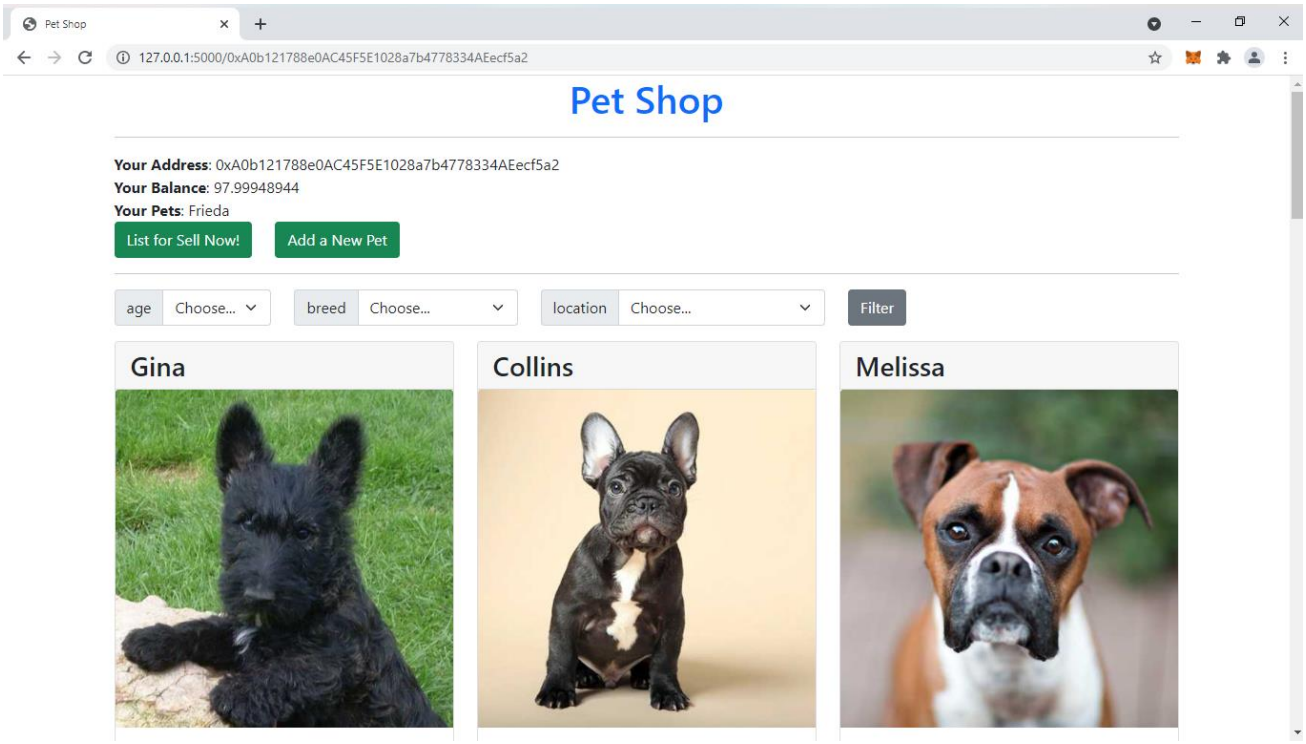


Figure 2 Account Information (screenshot)

Additionally, when the user clicks the 'List for Sell Now!' button, it redirects to the seller information page, which lists the pets owned by the seller in more detail.

2.2 Filter Based on Pet Attributes

All the pages that display the list of pets as cards offer the filter bar. It includes the dropdown button to select the criteria. The 'Filter' button submits the request to the back-end, which updates the pet cards displayed on the screen to match the criteria. The image below shows the options to choose from for the 'breed' criteria.

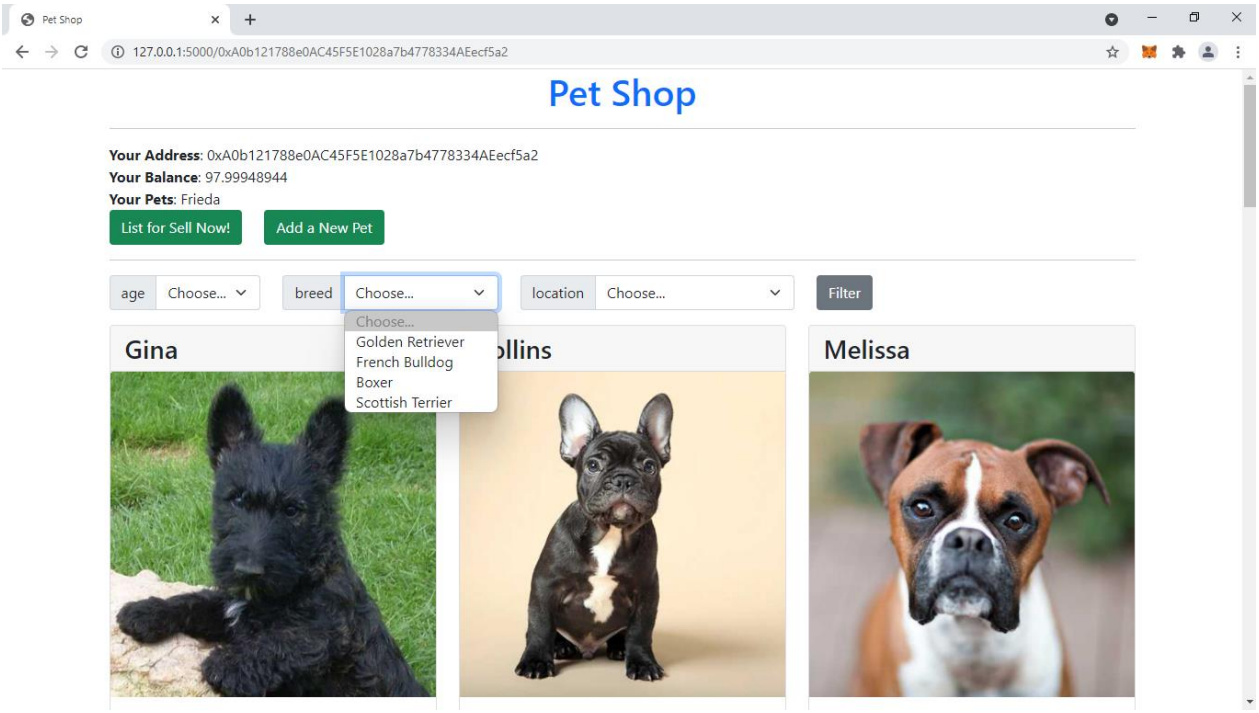


Figure 3 Filter Criteria for Attribute Breed (screenshot)

2.3 Buy a Pet

The section below the account information lists the pets up for sale. The user can filter them according to the attribute values, as explained in Section 2.2. All the pet cards display the following information in order: Name, Image, Breed, Age, Location, Price, and 'Buy' button. The image below illustrates this.

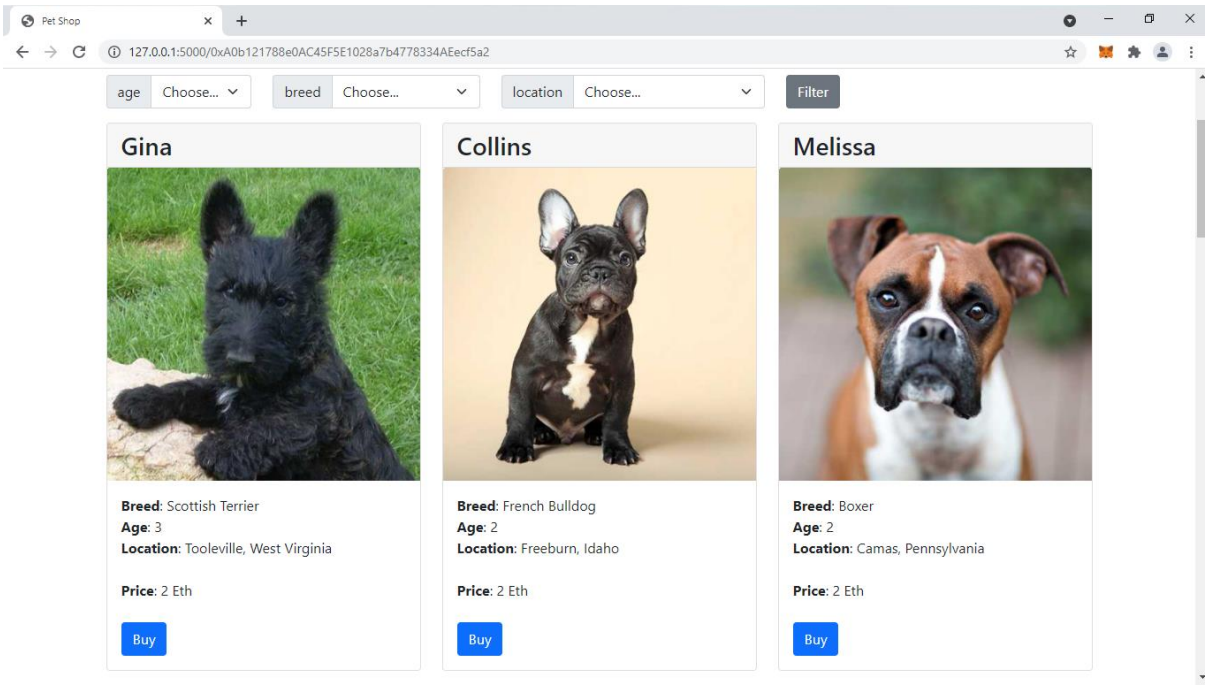


Figure 4 Pet Card (screenshot)

Please note that the ‘Buy’ button is clickable because user has selected the account, as can been seen in the URL. When the user clicks the ‘Buy’ button, it triggers the transaction using the selected account. The transaction transfers the amount specified in the ‘Price’ field on the pet card from the user’s account to the pet owner’s account. The transaction is facilitated by Web3 (web3.py). If the transaction fails, the user is redirected to the error page, where the stack trace is displayed. If it succeeds, the account information page is updated, and the pet card is removed from the list of pets up for sale.

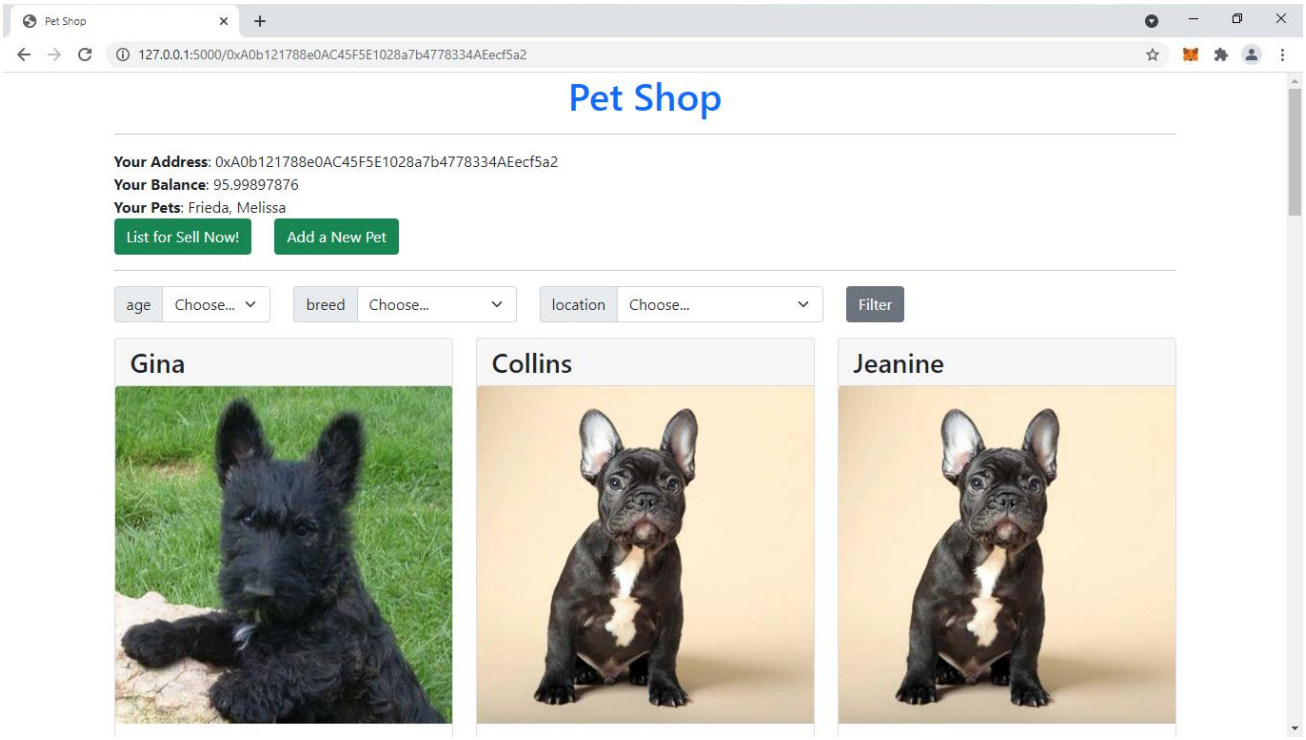


Figure 5 Information update after the successful buy (screenshot)

The above image shows the account information of the same user shown in figure 2 after purchasing pet ‘Melissa.’ Please note that the ‘Your Pets’ field is updated to include ‘Melissa’ and ‘Your Balance’ field is reduced after transferring the price to the previous owner.

2.4 Sell a Pet

When a user clicks on the ‘List for Sell Now!’ button, it is redirected to the seller information page. It looks like the one shown below, depending on the pets owned by the user. Please note how the list of pets is updated to show only the pet owned by the user. Also, note the text input box and the ‘List for Sell’ button added to each pet card.

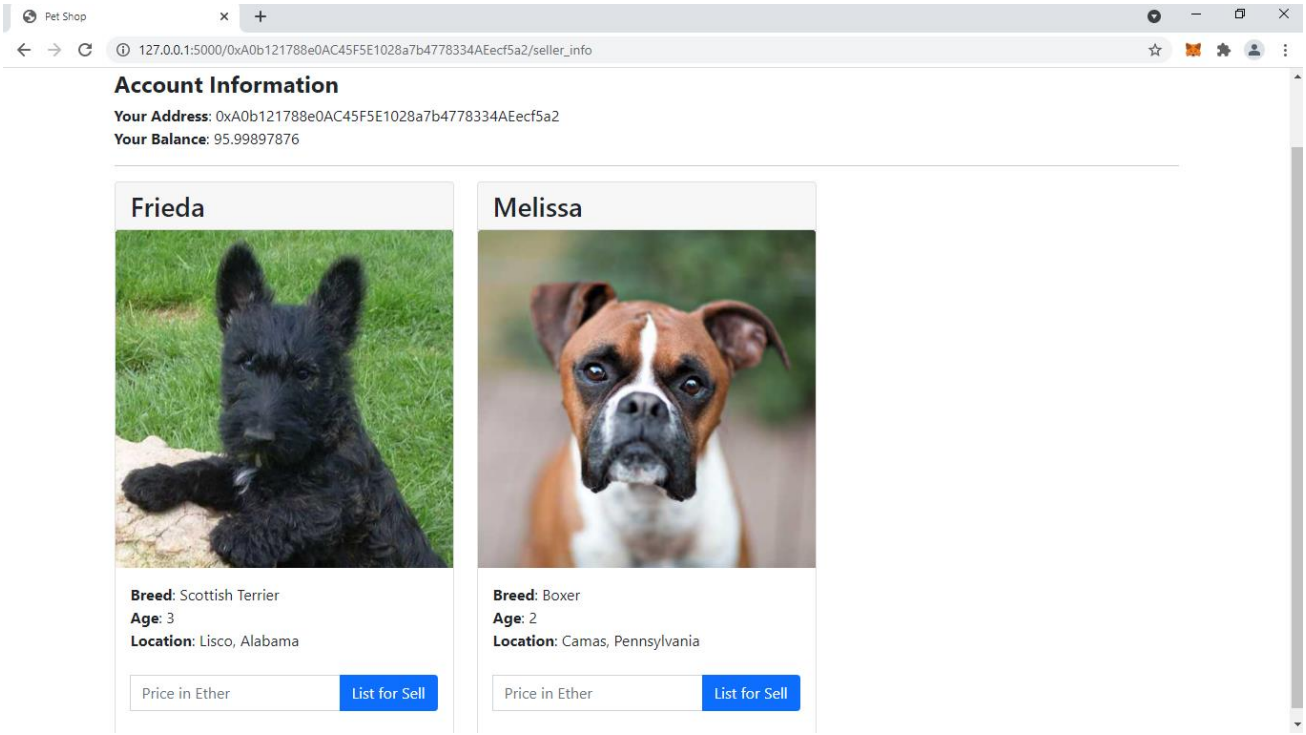


Figure 6 seller information page with options to list pets for sale (screenshot)

When the user enters the price and clicks the ‘List for Sell’ button, a transaction is triggered in the back-end using the Web3 to update the selling price of the pet and make it available for purchase. If the transaction succeeds, the seller's information is updated. Note that the pet is shown as ‘Listed for Sell’ with the selling price. The ownership is only transferred when someone buys the pet making a successful payment to the seller. This is shown in the image below.

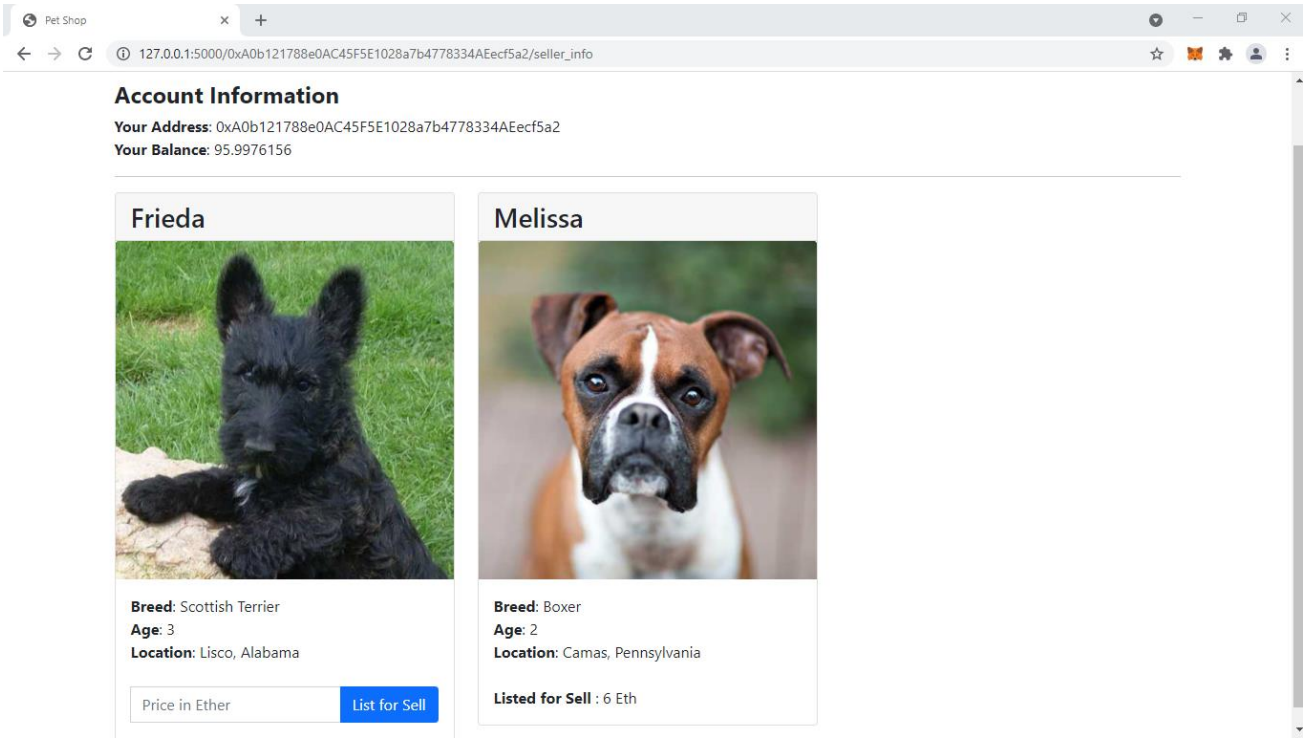


Figure 7 'Melissa' is listed for sale by the current owner (screenshot)

2.5 Add a Pet

When a user clicks on the 'Add a New Pet' button, it is redirected to the 'add pet' page. It looks like the one shown below. The page displays the input form with fields to collect the information about the pet from the owner. The fields include 'name,' 'age,' 'breed,' 'location,' and 'price.'

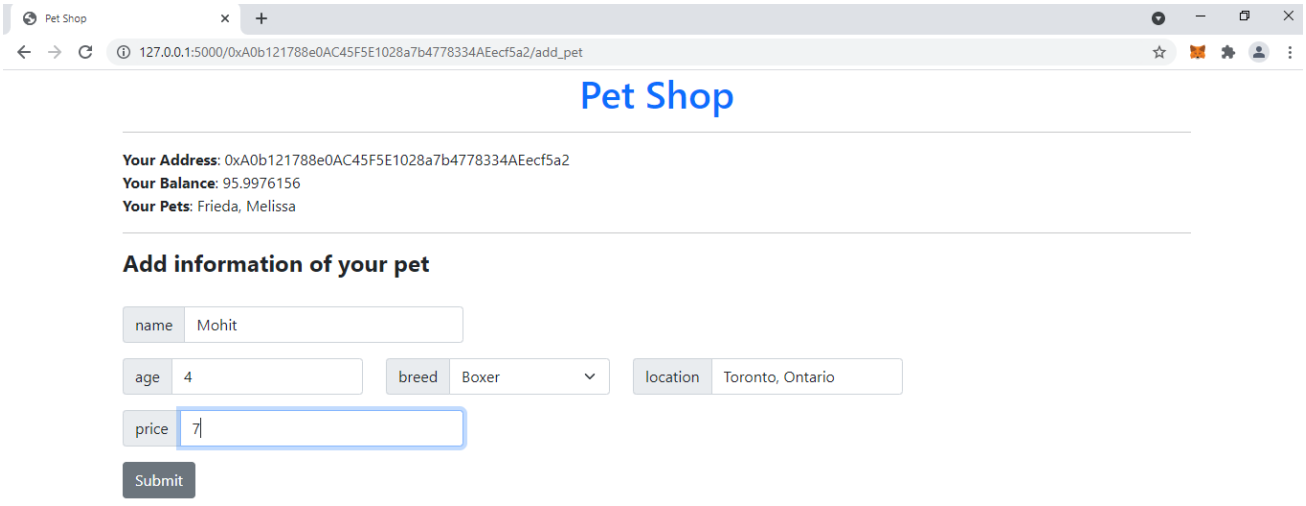


Figure 8 Form to collect information of new pet (screenshot)

When the user submits the form, a transaction is created from the user's account using Web3. If it succeeds, the user is registered as the owner. The pet is made available for purchase at a price provided by the owner. The pet is assigned a unique id in the contract on the blockchain. (what data about pets are stored on the blockchain is explained in the next section.). The image below shows the account status after adding a new pet.

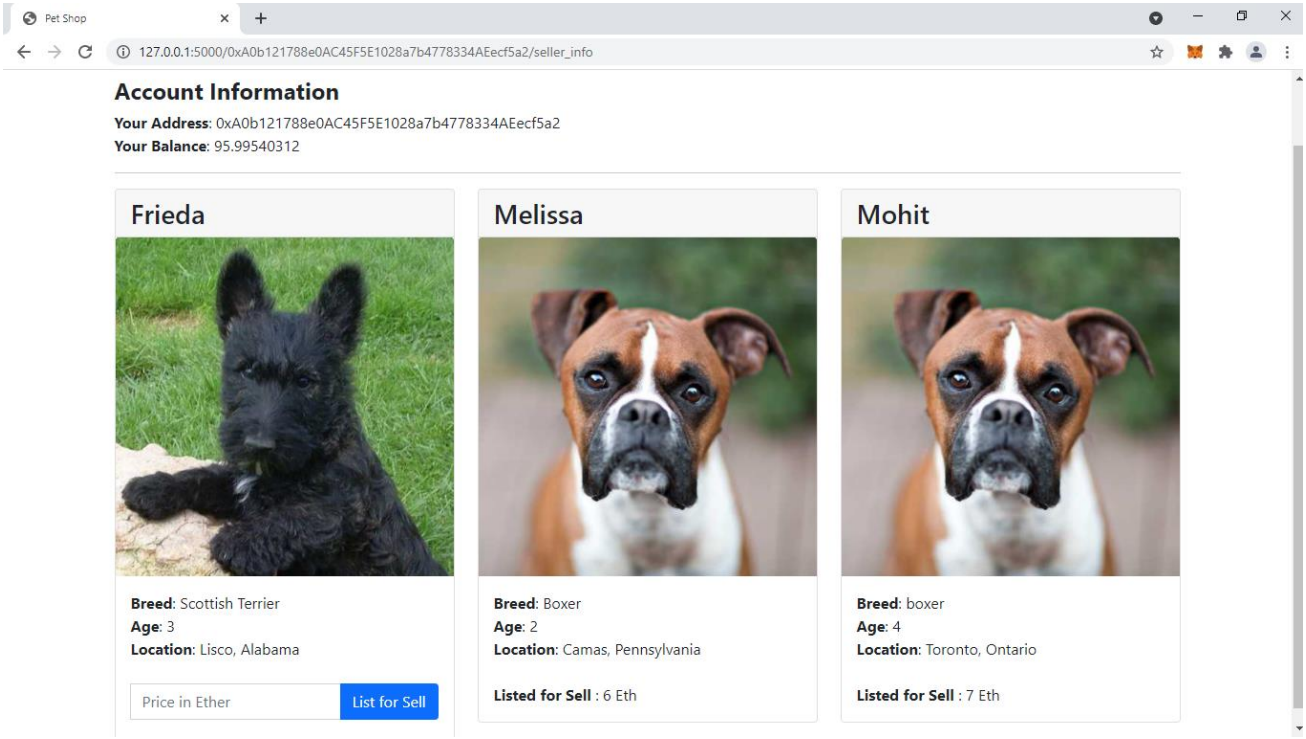


Figure 9 Account information of the user after adding 'Mohit' to the blockchain for sale (screenshot)

Additional minor features are illustrated in the video submitted with this form.

3 PetShop Smart Contract

The image below shows the directory structure of the project. The folders relevant to contract writing, compiling, and testing is expanded.

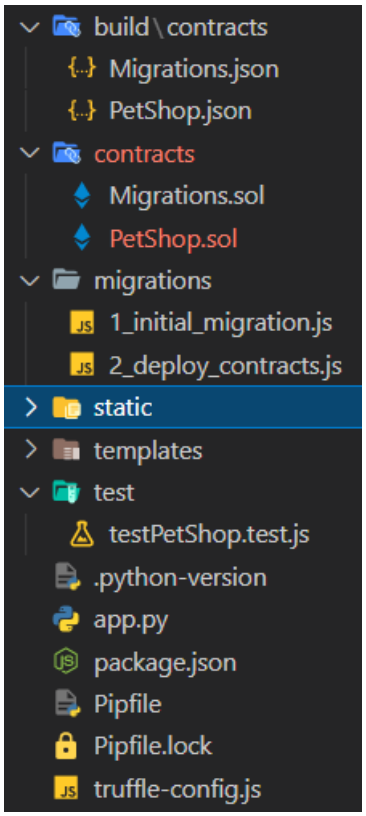


Figure 10 Directory structure (folder relevant to smart contract development are expanded)

The smart contract written in the Solidity file 'PetShop.sol' inside the 'contracts' folder is at the heart of the back-end logic of this application. After deployment to the blockchain, it acts as a distributed registry of the pet on the blockchain. At the same time, it allows anyone to modify the state of the registry according to the predefined rule that cannot be altered. It facilitates buying and selling transactions (change of ownership of the pets) without any intermediary. The code for the contract is explained in detail below.

3.1 Solidity Code

The image shows the first two lines of the contract code. Note that the file name is displayed at the top of the code box. The number on the left shows the line number in the file.

pet-shop-project - PetShop.sol

```
1  pragma solidity ^0.5.0;
2  pragma experimental ABIEncoderV2;
```

The first line is a directive for the compiler. It uses semantic versioning to indicate that the contract should be compiled using the compiler version 0.5 (any version between 0.5.0 to 0.6.0 but not including 0.6.0). The second line indicates that the code utilizes the experimental features introduced with the ABIEncoder2 and should be used with caution.)

pet-shop-project - PetShop.sol

```
4  contract PetShop {
5      struct Pet {
6          uint256 id;
7          address payable currentOwner;
8          uint256 price;
9          bool isForSell;
10     }
11
12     Pet[] public pets;
13
14     constructor(uint256 n) public payable {
15         // contract owner
16         for (uint256 i = 0; i < n; i++) {
17             pets.push(Pet(i, address(uint160(msg.sender)), 2 ether, true));
18         }
19     }
```

Line 4 marks the beginning of the contract code. The code block between lines 5 and 10 defines a struct ‘Pet’ to store information of a pet. It includes four fields: id, currentOwner, price, and isForSell. Their data types are marked. Note that the currentOwner is of type ‘address payable’. For owners to receive payment when they sell their pets, it is necessary to store addresses as payables.

Line 12 declares the array of type ‘Pet.’ It is a container datatype used to store all the pets.

Lines 14 to 19 define the constructor of the contract. It accepts the integer argument ‘n’ when the contract is deployed. ‘n’ is the number of pets to be included when is the contract is deployed. The constructor set the contract owner as the owner of the pets stored when deploying the contract. It also marks all the pets available for purchase.

pet-shop-project - PetShop.sol

```
21  // Retrieving the pets
22  function getPets() public view returns (Pet[] memory) {
23      return pets;
24  }
```

Next, Line 22 to 24 defines the function 'getPets'. It is a 'public' function of 'view' type and returns pets struct. Solidity categorizes functions in three types, as explained next.

3.1.1 Function types

1) **View Function:** functions that do not modify the state of the blockchain (that is, **do not write** to the blockchain via transaction) are categorized as 'view' functions.
2) **Pure Function:** functions that do not modify the state of the blockchain and do not read from the blockchain (that is, **do not read and write** to the blockchain via transaction) are categorized as 'pure' functions. Calling such functions does not cost any gas.
3) Functions not identified as one of the above can modify the state (read and write to blockchain).

3.1.2 Visibility

Contract attributes (functions and state variables) have one of the four types of visibility:

1) **Public:** visible to everyone
2) **Private:** visible inside contract
3) **Internal:** visible inside contract and child contract
4) **External:** visible only to other contracts and accounts

Next, the code shown below defines three modifiers: validateId, validateOwner, validatePrice.

3.1.3 Function Modifiers

Function modifiers are the reusable code that run before and after the function to which they are applied. They are mostly used for three use cases:

1) Restrict write access (by checking the owner): 'validateOwner' is an example of this use case.
2) Input Validation: 'validateId' and 'validatePrice' are examples of this use case.
3) Reentrancy Guard (not needed for PetShop contract)

pet-shop-project - PetShop.sol

```
26  modifier validateId(uint256 _id) {
27      require(_id >= 0 && _id < pets.length, "Invalid ID");
28      _;
29  }
30
31  modifier validateOwner(uint256 _id, address _owner) {
32      require(_id >= 0 && _id < pets.length, "Invalid ID");
33      Pet storage pet = pets[_id];
34      require(pet.currentOwner == _owner, "Not Owner!");
35      _;
36  }
37
38  modifier validatePrice(uint256 _price) {
39      require(_price >= 0 && _price < 2**256 - 1, "Invaid Price");
40      _;
41  }
```

All the modifiers end with '_;' statement (Line 28, 35, 40). The code before that line is executed when a function to which the modifiers are applied is called. The function code executes after that. If there is more code below '_;' statement, it would run after executing the function code before it returns.

Also, note that all the modifiers use the 'require' statement. It is one of the methods of error checking offered by Solidity. Error checking is discussed next.

3.1.4 Error Checking

Solidity offers three statements for error checking:

1) **Assert:** if the 'assert' condition fails, it consumes all the gas specified for the call.
2) **Require:** it doesn't use up all the gas and is most preferred for input validation, preconditions, and outputs.
3) **Revert:** It is similar to assert.

Next, the buy function is described. 'buy' function introduces one new concept: transfer of ether from one account to the other within a contract function. It is discussed next after the explanation of what the function does. 'buy' function takes the pet id as input and validates it using a modifier (Line 44). If id is valid, the function reads pet struct from the contract storage for the given id (Line 45). In Line 48 and 51, the code validates whether the pet is up for sale and the contract calling the

function has passed sufficient ether to pay for the price of the pet. ‘msg.value’ stores the amount transferred by the caller for this transaction. Next, on line 54, the code transfers the amount to the pet's current owner by calling the ‘transfer’ method on the ‘payable’ address of the owner. Line 57, 60, and 63 updates the pet's status, change the price to zero and set the function caller as a new owner of the pet.

pet-shop-project - PetShop.sol

```
43  // Buy a pet
44  function buy(uint256 _id) public payable validateId(_id) {
45      Pet storage pet = pets[_id];
46
47      // validate status
48      require(pet.isForSell == true, "Pet is not for sell");
49
50      // validate price
51      require(msg.value >= pet.price, "Insufficient Balance");
52
53      // send ether to current owner
54      pet.currentOwner.transfer(msg.value);
55
56      // update sell status
57      pet.isForSell = !pet.isForSell;
58
59      // update the price to zero
60      pet.price = 0;
61
62      // update the owner
63      pet.currentOwner = address(uint160(msg.sender));
64  }
```

3.1.5 Sending Ether

Solidity offers three methods to send ether.

1) Transfer: it forwards 2300 gas to function call and throws an error if the transaction fails.
2) Send: it also forwards 2300 gas but returns true (bool) if success or false (bool) if failure.
3) Call: forward all the gas received for the current function call and return bool like ‘send’.

In the ‘buy’ function, I used the ‘transfer’ method.

Next, the ‘listForSell’ function is described.

pet-shop-project - PetShop.sol

```
66  // list a pet for selling
67  function listForSell(uint256 _id, uint256 _price)
68      public
69      validateOwner(_id, msg.sender)
70  {
71      Pet storage pet = pets[_id];
72      if (pet.isForSell != true) {
73          pet.isForSell = !pet.isForSell;
74      }
75      pet.price = _price;
76  }
```

This function is called when a current owner of a pet wants to list it for selling. Line 68 validates whether a caller is the owner of a mentioned pet in the input argument using a modifier defined before. Next, code on line 71 retrieves the pet from the storage, line 73 mark it for sale by toggling the ‘isForSale’ boolean attribute if it is not already true, and finally, line 75 set the price.

Next, the ‘addPet’ method is described.

pet-shop-project - PetShop.sol

```
78  // add a pet
79  function addPet(uint256 _price)
80      public
81      validatePrice(_price)
82      returns (uint256)
83  {
84      uint256 id = pets.length;
85      pets.push(Pet(id, address(uint160(msg.sender)), _price, true));
86      return id;
87  }
```

First, the code on line 81 checks if the price specified is valid using a modifier. Line 84 creates the id for a new pet. Then on line 85, the pet struct is created and added to the array of pets using the array’s ‘push’ method. Note that the caller's address is set to the owner after converting the address into a payable form. Finally, the id of the new pet is returned.

pet-shop-project - PetShop.sol

```
89      // get price for a given id
90      function getPrice(uint256 _id)
91          public
92          view
93          validateId(_id)
94          returns (uint256)
95      {
96          Pet storage pet = pets[_id];
97          uint256 price = pet.price;
98          return price;
99      }
100
101     // get the owner for a given pet
102     function getOwner(uint256 _id)
103         public
104         view
105         validateId(_id)
106         returns (address)
107     {
108         Pet storage pet = pets[_id];
109         address owner = pet.currentOwner;
110         return owner;
111     }
112 }
```

The code above shows the last two methods of the contract: ‘getPrice’ and ‘getOwner’. Both are public and have ‘view’ visibility. The first returns the price, and the second returns the current owner if the id supplied to the function call is valid.

That’s all the Solidity code for the contract, as included in the project directory attached with this report.

3.2 Testing

For testing, JavaScript code is used. Truffle framework used for this project provides the functionality to easily write and perform tests. For instance, the assertion library 'chai' and testing framework 'mocha' are imported into the test files. Truffle also injects the web3 object (web3.js).

The following is the code used for testing:

pet-shop-project - testPetShop.test.js

```
const PetShop = artifacts.require("PetShop");

contract("PetShop", (accounts) => {
    let petshop;
    let expectedBuyer;
    let expectedSeller;
    let expectedPrice;

    before(async () => {
        petshop = await PetShop.deployed();
    });

    describe("Buying a pet and retrieving account addresses", async () => {
        before("Buy a pet using accounts[1]", async () => {
            await petshop.buy(8, { from: accounts[1], value: 2000000000000000000 });
            expectedBuyer = accounts[1];
        });

        it("can fetch the address of an owner by pet id", async () => {
            const owner = await petshop.getOwner(8);
            assert.equal(owner, expectedBuyer, "The owner of the pet should be the second account.");
        });

        it("can fetch the collection of all pets", async () => {
            const pets = await petshop.getPets();
            assert.equal(pets[8].currentOwner, expectedBuyer, "The owner of the pet should be in the collection.");
        })
    });
```

pet-shop-project - testPetShop.test.js

```
describe("List a pet for selling and retrieving price", async () => {
    before("List a pet using accounts[1]", async () => {
        await petshop.listForSell(8, "1000000000000000000", { from: accounts[1] });
        expectedPrice = "10000000000000000000";
        expectedSeller = accounts[1]
    });

    it("can fetch the price of an pet set by owner", async () => {
        const price = await petshop.getPrice(8);
        assert.equal(price, expectedPrice, "The price of the pet should equal to the amount set by seller.");
    });

    it("can fetch the collection of all pets and get the status", async () => {
        const pets = await petshop.getPets();
        assert.equal(pets[8].isForSell, true, "The pet should be listed for selling.");
    })
});
```

pet-shop-project - testPetShop.test.js

```
48      describe("Buying a listed pet from the seller and checking the deposit", async () => {
49          before("Buy a pet using another account: accounts[3]", async () => {
50              oldBalance = await web3.eth.getBalance(accounts[1]);
51              await petshop.buy(8, { from: accounts[3], value: "1000000000000000000" });
52          });
53
54          it("can fetch the account balance of the owner (account[1]) and verify deposit", async () => {
55              newBalance = await web3.eth.getBalance(accounts[1]);
56              assert.equal(parseInt(newBalance), parseInt(oldBalance) + parseInt(expectedPrice), "The owner should recieve the listed price.");
57          });
58
59          it("can check the status of the sold pet", async () => {
60              const pets = await petshop.getPets();
61              assert.equal(pets[8].isForSell, false, "The sold pet should change the status 'isForSell' to false.");
62          })
63
64          it("can fetch the address of new owner by pet id", async () => {
65              const newOwner = await petshop.getOwner(8);
66              assert.equal(newOwner, accounts[3], "The owner of the pet should be the fourth account.");
67          });
68      });
69  });
```

The tests are divided into three sections. The first part tests the buying functionality. The second part tests the listing functionality for selling. The third part tests the functionality of buying a listed pet. It also checks the ownership is transferred correctly and the amount is paid to the previous owner and deducted from the new owner's account.

4 Flask Application

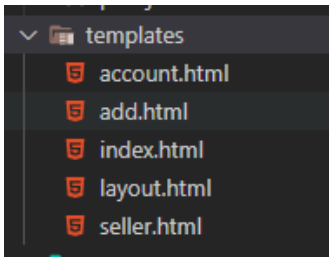
To build a user interface for the app, the python web-development framework Flask is used. Flask uses a Model-View-Controller (MVC) design pattern. It organizes the code into two directories and one additional file: the 'templates' directory is for all the HTML and CSS files; the 'static' directory is the place for all data files; the app.py file is the place where controller logic is implemented.

As such, the 'templates' directory corresponds to 'View', the 'static' directory corresponds to 'Model', and 'app.py' corresponds to 'Controller' in the MVC design pattern. The controller code written in 'app.py' listens for incoming requests and sends the appropriate responses back to the client. To serve the correct information, it dynamically generates the HTML files by using code logic, 'templates', and data stored in the 'static' directory. It handles all the routing and request processing as well.

The following sections briefly describe the files stored in each of the folders used by the framework.

4.1 templates (View)

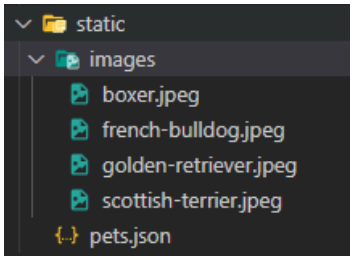
This folder contains the files shown in the image below:



'layout.html' is the base template from which all other HTML files inherit. Flask using ninja template language and engine to achieve this. The syntax of Jijna is very pythonic, beginner-friendly, and powerful at the same time. Depending on the client request, the app.py utilizes one of the templates and dynamically generates the HTML file to send to the user. For styling, all the HTML files use the Bootstrap library.

4.2 static (Model)

This folder contains the files shown in the image below:



The ‘pets.json’ stores the metadata about the pets, such as age, breed, location, etc. It identified the pets using the same id used in the solidity contract deployed on the blockchain. The ‘images’ directory stores the images of the pets.

4.3 app.py (Control)

This is the file where all the magic happens. When the ‘flask run’ command is executed, it runs the app.py file, which starts the server and starts listening for client requests. Depending on the client's request, it invokes one of the many decorated functions. The functions are decorated according to the route they are supposed to handle and return the response to the client. All this is standard in how the web application is built except web3 for contract deployed, transactions and calls.

So, next, interesting web3 code is highlighted in the snipped and briefly discussed.

4.3.1 Web3 (web3.py)

The web3.py is a python library that uses a JSON-RPC interface to communicate with Ethereum (test) blockchain. So whenever we need to read the data from the blockchain, make contract calls, or deploy the contract, the Web3 object from web3.py is used.

The following code deploys the PetShop contract compiled using truffle and stored in the ‘contracts’ directory. The comments explain what each line of code does. It finally stores the address of the deployed contract into the ‘contract_address’ variable, and the contract object in the ‘deployed_contract’ variable for later use when a user interacts with the application.

pet-shop-project - app.py

```python
# compile your smart contract with truffle first
with open("./build/contracts/PetShop.json") as f:
    truffleFile = json.load(f)

# web3.py instance
w3 = Web3(Web3.HTTPProvider("http://localhost:7545/"))

abi = truffleFile["abi"]
bytecode = truffleFile["bytecode"]

# set pre-funded account as sender
w3.eth.default_account = w3.eth.accounts[0]

# Instantiate and deploy contract
adopt_contract = w3.eth.contract(abi=abi, bytecode=bytecode)
# Get transaction hash from deployed contract
tx_hash = adopt_contract.constructor(16).transact()
# Get tx receipt
tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
# get contract address
contract_address = tx_receipt.contractAddress

# Contract instance in concise mode
deployed_contract = w3.eth.contract(abi=abi, address=contract_address)
```

The following web3 code is executed when a user buys a pet.

pet-shop-project - app.py

```python
def handle_buy(buyer, pet_id):
    try:
        # get the price of the pet
        price = deployed_contract.functions.getPrice(int(pet_id)).call()

        # print("price:", price)

        # make transaction from buyer's account
        deployed_contract.functions.buy(int(pet_id)).transact(
            {"from": buyer, "value": price}
        )

        update_pets_status()

        return "success"

    except Exception as e:
        return str(e)
```

The following code is executed when a user lists a pet for sale.

pet-shop-project - app.py

```
134  def handle_sell(seller, pet_id, price):
135      try:
136
137          # make transaction from seller's account
138          deployed_contract.functions.listForSell(
139              int(pet_id), w3.toWei(int(price), "ether")
140          ).transact({"from": seller})
141
142          update_pets_status()
143
144          return "success"
145
146      except Exception as e:
147          return str(e)
```

All the code and data files are available in the project folder attached with this report.

5 Conclusion

In this project, I demonstrated the use of blockchain for distributed e-commerce, which doesn't rely on the intermediary and is peer-peer. While I believe that the user interface is good enough for use, it can be improved significantly. The delays in response to the JSON-RPC calls made by Web3 when interacting with the contract and the blockchain, in general, creates significant challenges for both the interface design and back-end logic. On the front-end, it introduces a 'glitch' in the UX. On the back-end, the transactions are only confirmed after several blocks are mined on top of the blocks containing the transactions. This limits the use of such technology to specific categories of commerce that can tolerate delays in confirmation. It is expected that such delays will be minimized in the future with improvements in technology. At which point, I believe that blockchain-based day-to-day commerce will become a reality. At this point, there are still significant opportunities to develop e-commerce applications for categories that can tolerate the delays. The clever user interface designs can extend the use to several more categories.

Appendix

1) Project directory for DApp: Pet Shop
2) README file that provides the detailed instructions to set up and run the app.
3) A video demonstrating the use of this DApp
4) Executive Summary (attached separately)