=== Buy N’ Get ===

GitHub Repository Link

== Group Details ===

Group Number :- 49

Team Members :-

Abhinav Kumar Sinha - 2020012 (CSE)
Janhavi Bayanwar - 2020043 (CSE)
Prerak Semwal - 2020105 (CSE)
Vineet Kaul - 2020153 (CSE)

Contributions :-

Relational Schema	:	*Equal participation*
ER Diagram	:	*Equal Participation*
Data population (Python)	:	Prerak Semwal, Vineet Kaul
Data Population (SQL)	:	*Equal Participation*
SQL Queries	:	*Equal Participation*
MidSem Optimization	:	Prerak Semwal *(Debugging Assistance* - **Vineet Kaul**)
Triggers	:	Abhinav Kumar Sinha, Vineet Kaul, Prerak Semwal
View, Grants, Roles	:	Vineet Kaul, Prerak Semwal
Indexing, Query Optimization	:	Vineet Kaul, Janhavi Bayanwar, Prerak Semwal
UI Skeleton	:	Janhavi (created all Frames) Vineet Kaul and Prerak Semwal (Aesthetics)
UI Code Part	:	Prerak Semwal (*Debugging Assistance* - **Vineet Kaul**)

How did we collaborate ?

Each member would join the online group meeting and would collaborate virtually.

Most of the time, we would work on the same thing while discussing ideas and brainstorming,

Sometimes, we also compartmentalize the work.

=== Problem Statement ===

To implement a replica of a real world online retail store system which allows its users to buy/sell products with the store system acting as an intermediary, using concepts of Database and Management Systems, MySql and Python Programming Language.

=== Scope of Project ===

Project **BuynGet** is aimed to replicate an online retail store system, with a user-friendly and easy-to-learn User Interface.
The primary aim of the project is advanced towards implementing a good relational schema, table relations and optimized SQL queries along with user-friendly interface and integrated PL/SQL.
BuynGet allows its customers to buy/sell goods by acting as an efficient intermediary.
Note that, BuynGet Services are only available in The United States of America, due to the current limitations in procuring valid details about our customers.
*(**Disclaimer**: Currently we use Mockaroo for Data Population)*

Also, BuynGet‘s database ensures that details of those Orders which are completed are removed.
We store the delivery date in YYYY-MM-DD which won’t specify the exact time of the day when it was ordered.

=== Stakeholders ===

BuynGet primarily has two categories of Stakeholders :-

1. Owners
2. Customers
    a. Sellers
    b. Buyers

Owners :

Owners are the prime stakeholders of **BuynGet**'s Database. They have unrestricted privileges and access to all the data **except the user's** private credentials.

Customers :

Customers are all user who have already signed up for **BuynGet** services and therefore can now surf through the application.

Sellers :

Sellers are those customers who had sold products to **BuynGet** stock at least once since their sign-up.
Hence, *all sellers are customers but all customers need not be sellers.*

Buyers **:**

Buyers are those customers who had bought products from **BuynGet** stock at least once. Hence, *all buyers are customers but all customers need not be buyers.*

[NOTE] : - A customer may be both buyer and seller.

=== ER Diagram ===

[Miro Link to ER Diagram]

=== Tables ===

owners

Field	Type	Null	Key	Default	Extra
owner_id	int	NO	PRI	NULL	
owner_name	varchar(50)	NO		NULL	
email	varchar(50)	NO	UNI	NULL	
password	varchar(50)	NO		NULL	
phone_no	varchar(20)	NO	UNI	NULL	

accounts

Field	Type	Null	Key	Default	Extra
username	varchar(50)	NO	PRI	NULL	
email	varchar(50)	NO	UNI	NULL	
password	varchar(50)	NO		NULL	
customer_id	int	NO	UNI	NULL	auto_increment

customers

Field	Type	Null	Key	Default	Extra
customer_id	int	NO	PRI	NULL	auto_increment
customer_name	varchar(50)	NO		NULL	
age	int	YES		NULL	
gender	varchar(50)	YES		NULL	
phone_no	varchar(50)	NO		NULL	
country	varchar(50)	NO		NULL	
state	varchar(50)	NO		NULL	
street_name	varchar(50)	NO		NULL	
street_no	int	NO		NULL	
pincode	varchar(5)	NO		NULL	

buyers

Field	Type	Null	Key	Default	Extra
customer_id	int	NO	PRI	NULL	

sellers

Field	Type	Null	Key	Default	Extra
customer_id	int	NO	PRI	NULL	

sells

Field	Type	Null	Key	Default	Extra
customer_id	int	NO	MUL	NULL	
item_id	int	NO	MUL	NULL	

items

Field	Type	Null	Key	Default	Extra
item_id	int	NO	PRI	NULL	auto_increment
item_name	varchar(50)	NO		NULL	
item_type	varchar(20)	NO		NULL	
quantity	int	YES		NULL	
cost_price	float	NO		NULL	
selling_price	float	NO		NULL	

carts

Field	Type	Null	Key	Default	Extra
cart_id	int	NO	PRI	NULL	auto_increment
customer_id	int	NO	UNI	NULL	

stores

Field	Type	Null	Key	Default	Extra
cart_id	int	NO	MUL	NULL	
item_id	int	NO		NULL	
quantity	int	NO		NULL	

orders

Field	Type	Null	Key	Default	Extra
order_id	int	NO	PRI	NULL	auto_increment
customer_id	int	NO	MUL	NULL	
order_date	date	NO		NULL	
delivery_date	date	NO		NULL	

ordered_items

Field	Type	Null	Key	Default	Extra
order_id	int	NO	MUL	NULL	
item_id	int	NO	MUL	NULL	
quantity	int	YES		NULL	

transactions

Field	Type	Null	Key	Default	Extra
order_id	int	NO	PRI	NULL	
customer_id	int	NO	MUL	NULL	
mode	varchar(20)	NO		NULL	
amount	int	NO		NULL	

payments

Field	Type	Null	Key	Default	Extra
payment_id	int	NO	PRI	NULL	auto_increment
customer_id	int	NO	MUL	NULL	
mode	varchar(20)	NO		NULL	
amount	int	NO		NULL	

feedback

Field	Type	Null	Key	Default	Extra
customer_id	int	NO	PRI	NULL	
rating	int	YES		NULL	

support

Field	Type	Null	Key	Default	Extra
support_id	int	NO	PRI	NULL	auto_increment
customer_id	int	YES	MUL	NULL	
issue	varchar(1000)	YES		NULL	
issue_date	date	YES		NULL	

=== SQL Queries (Optimized + Indexes) ===

1. Identify the most common mode of Payment OR Transaction.

SELECT DISTINCT(P.mode) AS 'Common Mode of Payments among Sellers' FROM payments AS P WHERE (SELECT COUNT(*) FROM payments WHERE mode = P.mode) IN (SELECT MAX(modeCount) FROM (SELECT COUNT(*) AS modeCount FROM payments GROUP BY mode) modeCount);

Optimization :-

- create index for *'mode'* column of table payments
- Avoid use of WHERE clause
- Using DESC LIMIT instead of using DISTINCT and COUNT(*)

CREATE INDEX mode_ ON payments (mode);

SELECT mode AS 'Common Mode of Payments among Sellers' FROM payments USE INDEX (mode_) GROUP BY mode ORDER BY count(mode) DESC LIMIT 1;

SELECT DISTINCT(T.mode) AS 'Common Mode of Transactions among Buyers' FROM transactions AS T WHERE (SELECT COUNT(*) FROM transactions WHERE mode = T.mode) IN (SELECT MAX(modeCount) FROM (SELECT COUNT(*) AS modeCount FROM transactions GROUP BY mode) modeCount);

Optimization :-

- create index for *'mode'* column of table payments
- Avoid use of WHERE clause
- Using DESC LIMIT instead of using DISTINCT and COUNT(*)

CREATE INDEX mode__ ON transactions (mode);

SELECT mode AS 'Common Mode of Payments among Buyers' FROM transactions USE INDEX (mode__) GROUP BY mode ORDER BY count(mode) DESC LIMIT 1;

2. Identify those customers who have ordered from at least three categories.

```
SELECT DISTINCT(O1.customer_id) FROM orders as O1 WHERE (SELECT
COUNT(DISTINCT(item_type)) FROM items) = (SELECT
COUNT(DISTINCT(item_type)) FROM items WHERE item_id IN (SELECT item_id
FROM orders WHERE customer_id = O1.customer_id));
```

---

3. Identify the total number of sales in each category.

```
SELECT items.item_type as 'Category', sum(ordered_items.quantity) AS 'Sales' FROM
ordered_items, items WHERE ordered_items.item_id = items.item_id  AND
item_type='Grocery' UNION SELECT items.item_type, sum(ordered_items.quantity)
FROM ordered_items, items WHERE ordered_items.item_id = items.item_id  AND
item_type='Electronics'
UNION SELECT items.item_type, sum(ordered_items.quantity) FROM ordered_items,
items WHERE ordered_items.item_id = items.item_id  AND item_type='Daily care';
```

Optimization :-

- using INNER JOIN instead of WHERE clause (as joins execute faster than where, however as per some online forums in some scenarios they are almost equivalent in terms of efficiency)

```
SELECT items.item_type AS 'Category', sum(ordered_items.quantity) AS 'Sales' FROM
ordered_items INNER JOIN items ON ordered_items.item_id = items.item_id  AND
item_type='Grocery' UNION SELECT items.item_type, sum(ordered_items.quantity)
FROM ordered_items INNER JOIN items ON ordered_items.item_id = items.item_id
AND item_type='Electronics'
UNION SELECT items.item_type, sum(ordered_items.quantity) FROM ordered_items
INNER JOIN items ON ordered_items.item_id = items.item_id  AND item_type='Daily
care';
```

---

4. Express the feedback ratings as a ratio measure.

```
select (select count(*) from feedback where rating >= (select avg(rating) from
feedback))/(select count(*) from feedback where rating < (select avg(rating) from
feedback)) AS 'Feedback Ratio';
```

5. Generate the current Male-to-Female ratio signed up in BuynGet's Database.

```
select ((select COUNT(*) from customers where gender = 'Male') / (select COUNT(*) from customers where gender = 'Female')) AS 'Male:Female Ratio';


CREATE INDEX gen ON customers(gender);

select ((select COUNT(*) from customers use index (gen) where gender = 'Male')/(select COUNT(*) from customers use index (gen) where gender = 'Female')) AS 'Male:Female Ratio';
```

---

6. Identify the most popular product/s among buyers.

```
SELECT item_id, item_name FROM items AS I1 WHERE ((SELECT COUNT(*) FROM ordered_items GROUP BY item_id HAVING item_id = I1.item_id) = (SELECT max(target) FROM (SELECT COUNT(*) AS target FROM ordered_items GROUP BY item_id) target));


CREATE INDEX order_access ON ordered_items (item_id);

SELECT item_id, item_name FROM items AS I1 WHERE ((SELECT COUNT(*) FROM ordered_items USE INDEX (order_access) GROUP BY item_id HAVING item_id = I1.item_id) = (SELECT max(target) FROM (SELECT COUNT(*) AS target FROM ordered_items USE INDEX (order_access) GROUP BY item_id) target));
```

---

7. List the buyers which are inactive (that is empty carts and no orders made yet).

```
SELECT username, email, customer_id FROM accounts WHERE customer_id IN (SELECT customer_id FROM carts WHERE cart_id NOT IN (SELECT cart_id from stores)) AND customer_id IN (SELECT customer_id FROM buyers WHERE customer_id NOT IN (SELECT customer_id FROM transactions));
```

Optimization :-

We can use INNER JOIN + UNION instead of WHERE clause but multiple joins probably would turn out to be more expensive task at server-end contrary to a WHERE clause

8. Product with the highest returns.

```
SELECT item_name FROM items WHERE selling_price - cost_price = (SELECT
max(selling_price - cost_price) FROM items);


CREATE INDEX prices ON items (item_name, cost_price, selling_price);

SELECT item_name FROM items USE INDEX (prices) WHERE selling_price - cost_price =
(SELECT  max(selling_price - cost_price) FROM items USE INDEX (prices));
```

---

9. Find those customers who are both seller and buyer.

```
SELECT customer_id FROM buyers WHERE customer_id in (SELECT
customer_id from sellers);
```

---

10. Find those customers who haven't ordered yet.

```
SELECT customer_id FROM buyers WHERE customer_id NOT IN (SELECT
customer_id FROM transactions);
```

---

11. Oldest unresolved support request.

```
SELECT customer_id FROM support WHERE issue_date <= ALL (SELECT issue_date
FROM support);
```

---

12. List down owner contact info.

```
SELECT owner_name, email, phone_no from owners;
```

Recap of MidSem Evaluation

Corrections Made (as per TA's remarks)

1. **Foreign Keys corrections - missing Foreign Keys pointed by the TA are included :**
   - declared customer_id as foreign key in : **payments , transactions , carts , feedback , support , sells**
   - declared order_id as foreign key in **transactions**
   - declared item_id as foreign key in : **orders , sells**
   - (**carts , sells**) tables now use (**buyers , sellers**) tables respectively for foreign key , *(this was just to make the schema more meaningful)*
2. **Many-to-Many Mapping issue pointed by TA fixed :**
   - **sells** table is created corresponding to m-n mapping between ***sellers - items***
   - ***stores*** table is created corresponding to m-n mapping between ***buyers - items***
3. The lines connecting the entities in ER-Diagram were having gaps for aesthetic purposes but as per TA's advice we fixed those and extended the lines.
4. We wrongly identified a proper entity as a weak entity **just for the sake of it**, as we thought it was compulsory to have one. This has also been fixed.
5. TA suggested that it would be a better approach to have only one order_id for all the items ordered in one go. This has also been taken care of.
6. TA's suggestion about cascading is also fulfilled.

Views

```sql
CREATE VIEW owners_info AS
SELECT owner_name AS Name, email
FROM owners;

CREATE VIEW accounts_info AS
SELECT username, email
FROM accounts;

CREATE VIEW customers_info AS
SELECT customer_name, gender, phone_no, country, state, street_name, street_no, pincode
FROM customers;

CREATE VIEW items_info AS
SELECT item_id, item_name, item_type, quantity, selling_price
FROM items;
```

Grants and Roles

```
CREATE role team;

GRANT SELECT ON owners_info to team;
GRANT SELECT ON customers to team;
GRANT DELETE ON accounts to team;
GRANT SELECT ON accounts_info to team;
GRANT SELECT ON sellers to team;
GRANT SELECT ON buyers to team;
GRANT ALL ON items to team;
GRANT SELECT ON orders to team;
GRANT SELECT ON ordered_items to team;
GRANT SELECT ON support to team;
GRANT SELECT, DELETE ON feedback to team;
GRANT SELECT ON payments to team;
GRANT SELECT ON transactions to team;
GRANT SELECT ON sells to team;

CREATE USER owner1@localhost identified by 'prerak';
CREATE USER owner2@localhost identified by 'vineet';
CREATE USER owner3@localhost identified by 'janhavi';
CREATE USER owner4@localhost identified by 'abhinav';

GRANT team TO owner1@localhost;
GRANT team TO owner2@localhost;
GRANT team TO owner3@localhost;
GRANT team TO owner4@localhost;
```

```
CREATE role buy;

GRANT SELECT ON owner_info to buy;
GRANT UPDATE ON customers_info to buy;
GRANT SELECT ON transactions to buy;
GRANT ALL ON support to buy;
GRANT SELECT ON items to buy;

CREATE user buyer1@localhost identified by 'buyer1';
CREATE user buyer2@localhost identified by 'buyer2';

GRANT buy TO buyer1@localhost;
GRANT buy TO buyer2@localhost;
```

```
CREATE role sell;

GRANT SELECT ON owner_info to sell;
GRANT UPDATE ON customers to sell;
GRANT SELECT ON payments to sell;
GRANT ALL ON support to sell;
GRANT SELECT ON items to sell;

CREATE user seller1@localhost identified by 'seller1';
CREATE user seller2@localhost identified by 'seller2';

GRANT buys TO seller1@localhost;
GRANT buys TO seller2@localhost;
```

Triggers

```
# if new rating entry inserted is less than the average, automatically issue a support entry
delimiter //
CREATE TRIGGER low_rating AFTER INSERT ON feedback
   FOR EACH ROW
   BEGIN
   DECLARE new_rating int;
   DECLARE Cid int;
   SET new_rating = NEW.rating;
   SET Cid = NEW.customer_id;

               IF new_rating < (SELECT AVG(rating) FROM feedback) THEN
                       INSERT INTO support (customer_id, issue, issue_date) values (Cid, 'Not Happy with Buynget Services', current_date());
               END IF;
   END;//
delimiter ;
```

```
# after each order, update the stock
delimiter //
CREATE TRIGGER stock_updation AFTER INSERT ON ordered_items
   FOR EACH ROW
   BEGIN
   DECLARE id int;
   DECLARE q int;
   SET id = NEW.item_id;
   SET q = NEW.quantity;

   UPDATE items SET items.quantity = items.quantity - q where item_id = id;

   END;//
delimiter ;
```

```
# delete all those order-entries from 'orders' table which are completed
delimiter //
CREATE TRIGGER order_completed BEFORE INSERT ON carts
   FOR EACH ROW
   BEGIN
               DELETE FROM orders where delivery_date < current_date();
   END;//
delimiter ;
```

```
# if age is NULL replace with 18
delimiter //
CREATE TRIGGER insertion BEFORE INSERT ON customers
   FOR EACH ROW
   BEGIN
           IF NEW.age IS NULL THEN
                   SET NEW.age = 18;
           END IF;
   END;//
delimiter ;
```

Embedded SQL Queries

- If a seller tries to sell an item which s/he did not had in the preference list than we ask the seller if s/he wishes to add that particular item to preference list or cancel the sale of that product.

```
def add_for_sell():
    item_id  = int(variables[15].get())
    quantity = int(variables[16].get())

    myCursor.execute("select item_id from sells where customer_id = {}".format(USER_ID))
    allowed_items = myCursor.fetchall()
    temp = []
    for x in allowed_items:
        temp.append(x[0])
    allowed_items = temp
    if item_id not in allowed_items:
        verdict = MessageBox.askquestion("Alert", "Trying to sell something new ! Want to add this item to your preference ?")
        if(verdict == "yes"):
            myCursor.execute("insert into sells values ({}, {})".format(USER_ID, item_id))
            myDataBase.commit()

            add = []
            add.append(item_id)
            add.append(quantity)
            variables[17].append(add)
    else:
        add = []
        add.append(item_id)
        add.append(quantity)
        variables[17].append(add)
```

- Here we try to replicate the sign-up process, where we must keep a check if entered data like username and email are unique or not while creating an account for a new user.

```
def sign_up():
    myCursor.execute("select count(*) from accounts where username = '{}'".format(sign_up_data[0]))
    count1 = myCursor.fetchall()
    count1 = count1[0][0]
    myCursor.execute("select count(*) from accounts where email = '{}'".format(sign_up_data[1]))
    count2 = myCursor.fetchall()
    count2 = count2[0][0]
    if count1 == 1 or count2 == 1:
        if count1 == 1:
            MessageBox.showinfo( "Alert", "Username already taken !")
            variables[3].set("")
        else:
            MessageBox.showinfo( "Alert", "Email already taken !")
            variables[4].set("")

    else:
        myCursor.execute("insert into accounts (username, email, password) values ('{}', '{}', '{}')".format(sign_up_data[0], sign_up_d
        myDataBase.commit()


        myCursor.execute("select count(*) from accounts")
        Cid         = myCursor.fetchall()
        Cid         = Cid[0][0]
        myCursor.execute("insert into customers(customer_id, customer_name, age, gender, phone_no, country, state, street_name, street_
        myDataBase.commit()
#-----------------------------------------------------------------------------------------------------------------------------------
```

- To allow empty cart functionality, if the user wishes to drop the items in the cart. Note that, we have to deal with the error-exception case if cart is already empty

```
def empty_cart():
    variables[20].clear()
    myCursor.execute("select cart_id from carts where customer_id = {}".format(USER_ID))
    cart_id = myCursor.fetchall()
    try:
        cart_id = cart_id[0][0]
        myCursor.execute("delete from stores where cart_id = {}".format(cart_id))
        myDataBase.commit()
    except:
        pass
```

- Here we check whether a user trying to log in to BuynGet has an existing account or not

```
def login():
    global USER_ID
    username = variables[0].get()
    password = variables[1].get()

    myCursor.execute("select * from accounts where username = '{}' and password = '{}'".format(username, password))
    count = myCursor.fetchall()

    try:
        count    = count[0]
        USER_ID = count[3]
        MessageBox.showinfo( "Logged In", "Welcome " + username + " !")
        variables[2].pack_forget()
        category()
    except:
        variables[0].set("")
        variables[1].set("")
        MessageBox.showinfo( "Alert", "Invalid Credentials")
```

- We use PI/SQL to populate the *'order table'* and the *'ordered_items'* table with newly placed orders by using python to do some calculations.

```
def place_order():
    myCursor.execute("insert into orders (customer_id, order_date, delivery_date) values ({}, current_date(), current_date())".format(USER_ID))
    myDataBase.commit()

    myCursor.execute("select count(*) from orders")
    order_id = myCursor.fetchall()
    order_id = order_id[0][0]
    for item_ in variables[20]:
        item               = item_[0]
        item_id            = item[0]
        item_name          = item[1]
        item_selling_price = float(item[2])
        item_quantity      = item[3]
        variables[21] += item_quantity * item_selling_price
        # we don't allow quantity = 0, SQL constraint will catch it
        myCursor.execute("insert into ordered_items values ({}, {}, {})".format(order_id, item_id, item_quantity))
        myDataBase.commit()
#-----------------------------------------------------------------------------------------------------------------------------------
```

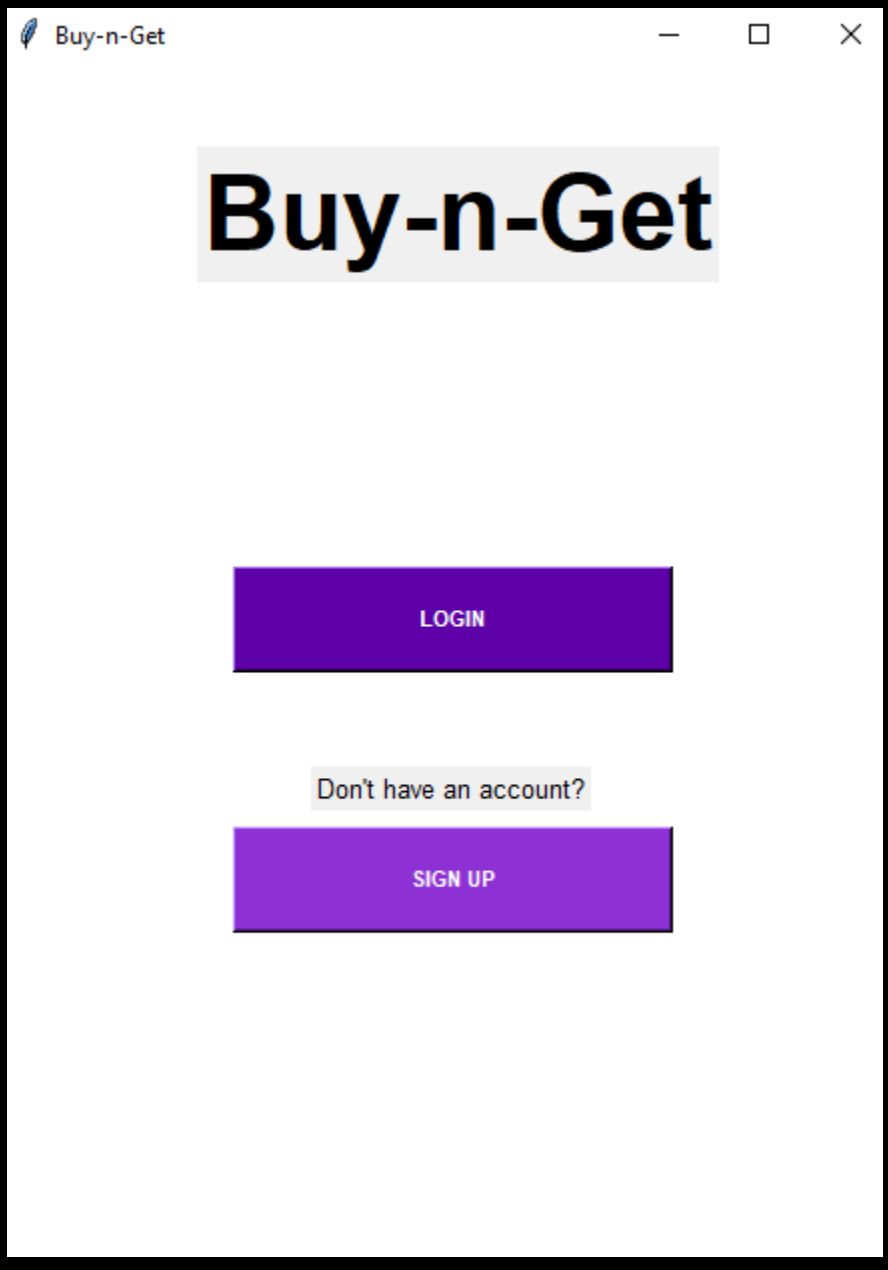
- We make sure that if a customer opts to buy on BuynGet platform then s/he must be present in buyers table

```
def buyer_page():
    try:
        myCursor.execute("insert into buyers values({})".format(USER_ID))
        myDataBase.commit()
        myCursor.execute("insert into carts (customer_id) values({})".format(USER_ID))
        myDataBase.commit()
    except:
        pass
    buyer_page_frame    = Frame(window, width=450, height=600, bg = "#ffffff")
```

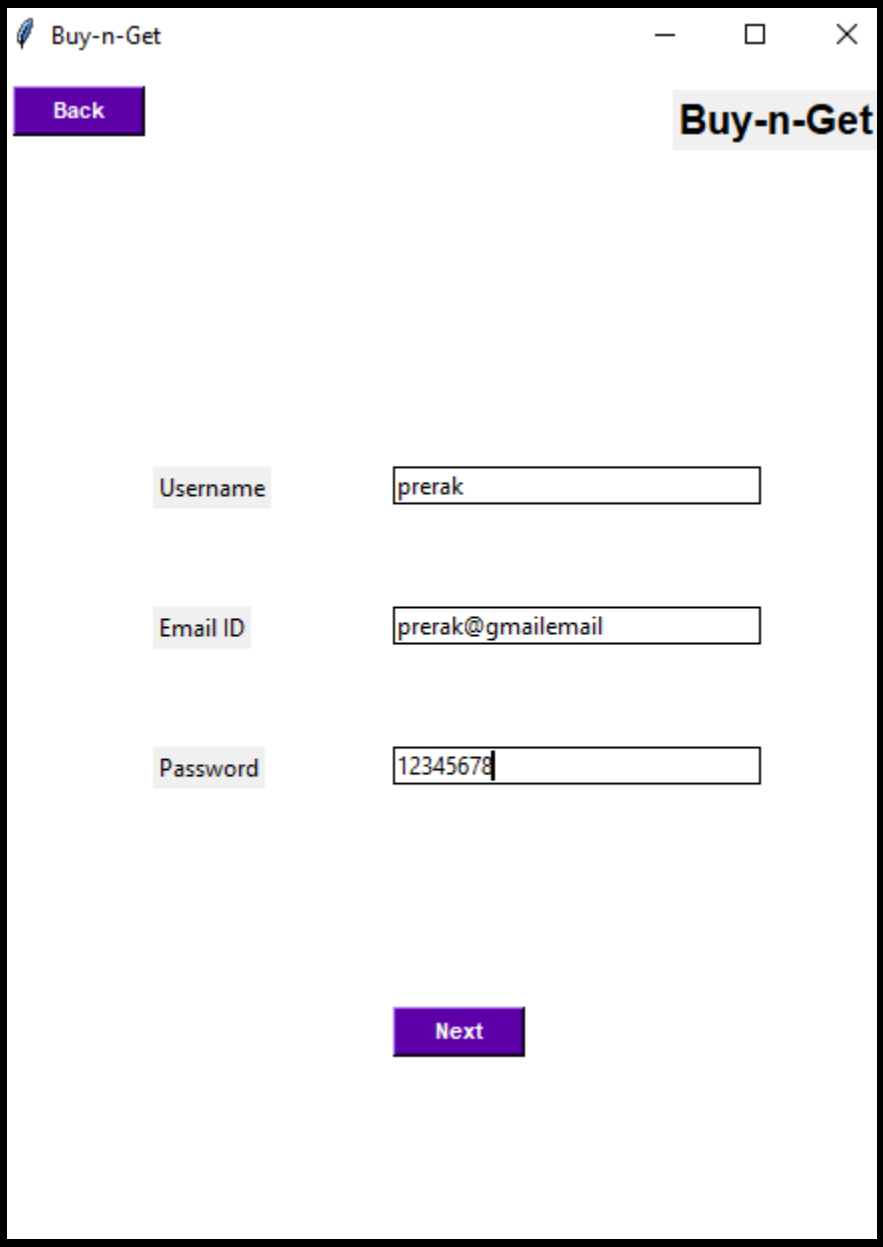
- We make sure that if a customer opts to sell on BuynGet platform then s/he must be present in sellers table

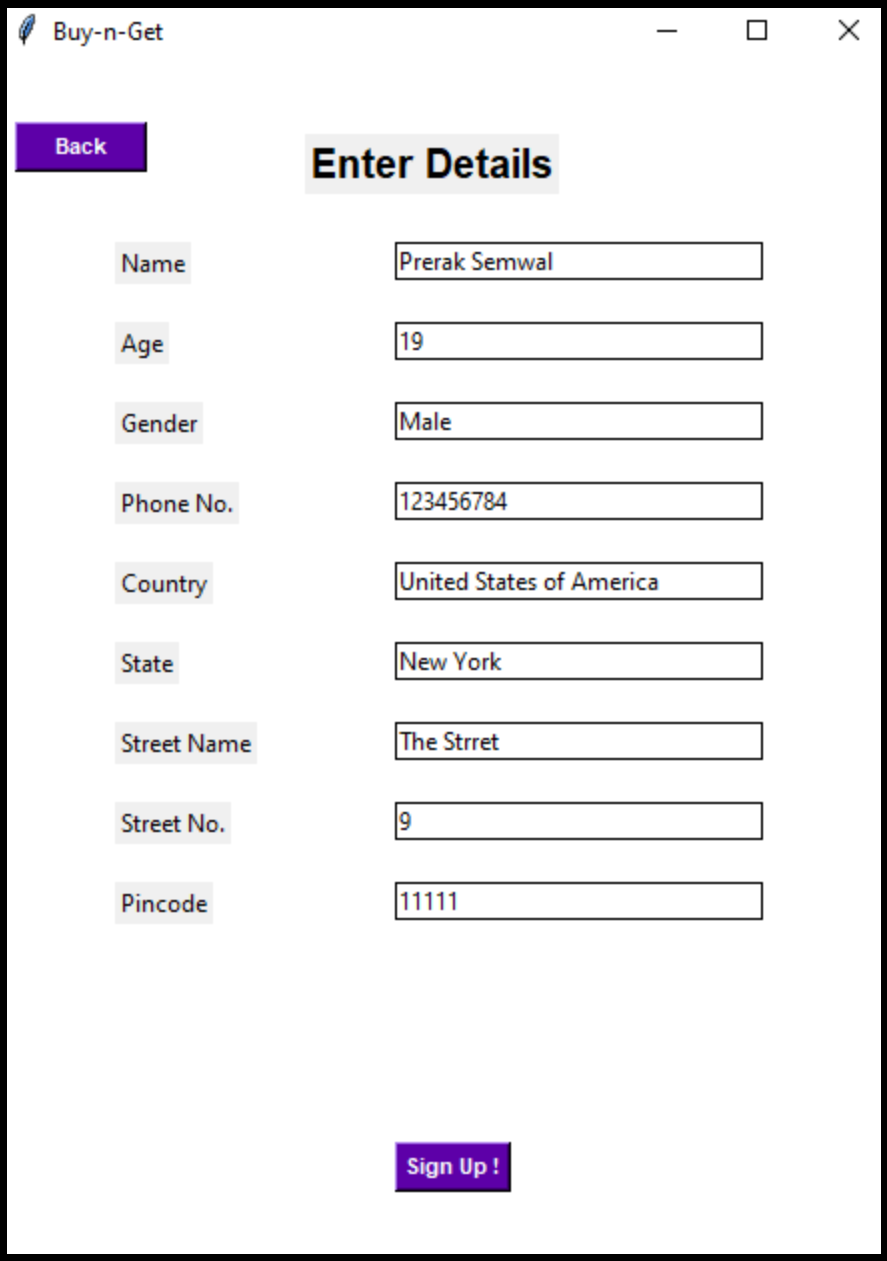
```
def seller_page():
    try:
        myCursor.execute("insert into sellers values({})".format(USER_ID))
        myDataBase.commit()
    except:
        pass

    seller_page_frame    = Frame(window, width=450, height=600, bg = "#ffffff")
```

UI Screenshots

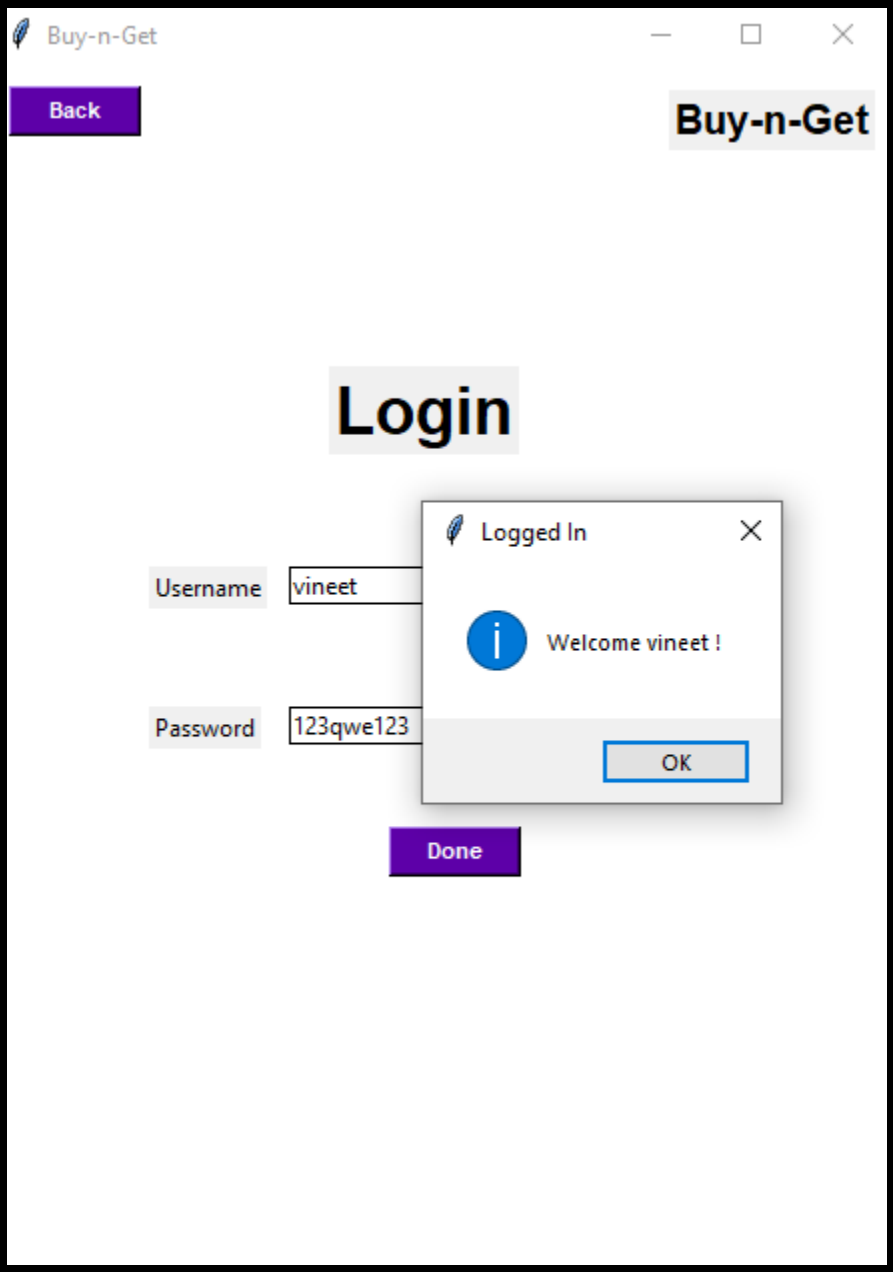


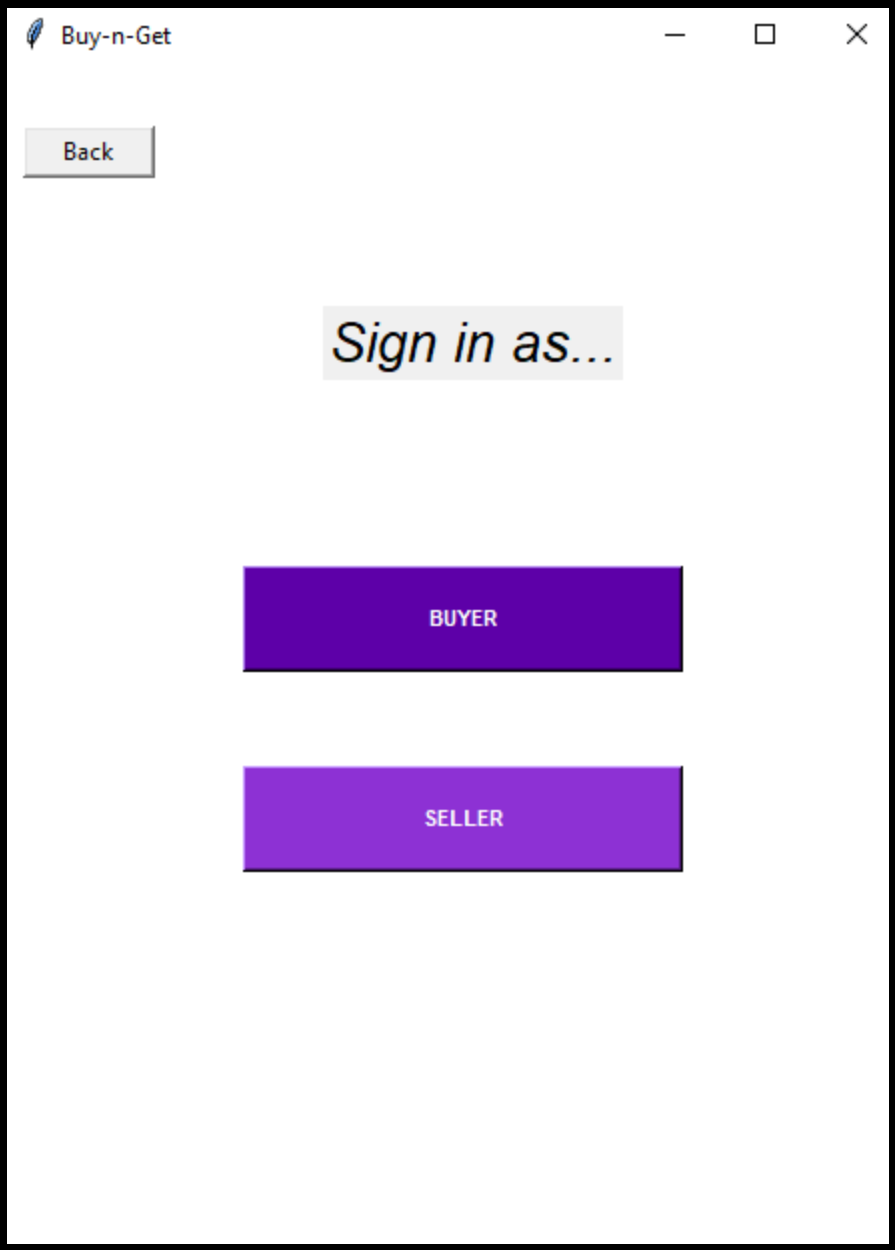
Sign Up


Buy-n-Get
Back
Buy-n-Get
Username
prerak
Email ID
prerak@gmailemail
Password
12345678
Next

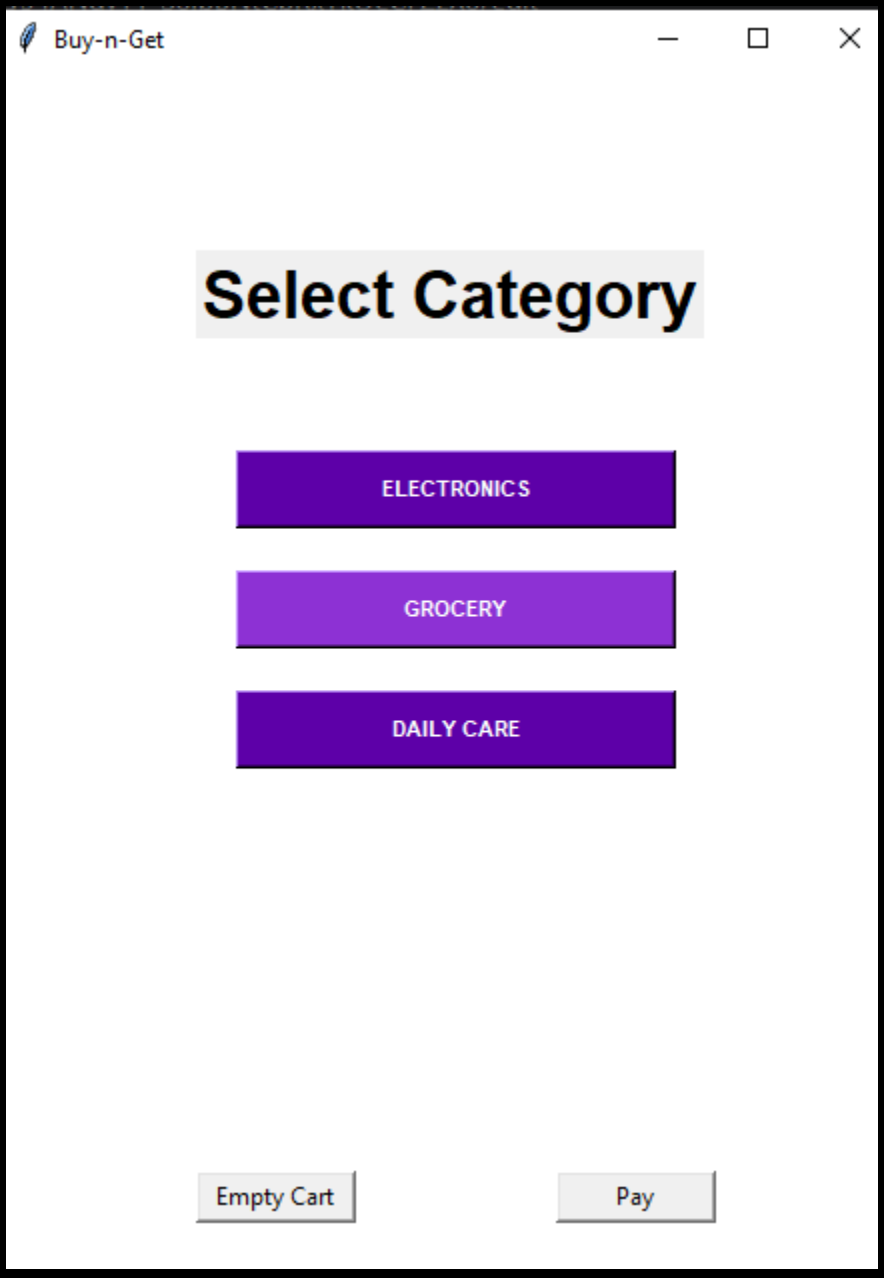

Buy-n-Get
Back
Enter Details
Name
Prerak Semwal
Age
19
Gender
Male
Phone No.
123456784
Country
United States of America
State
New York
Street Name
The Strret
Street No.
9
Pincode
11111
Sign Up !

Log In

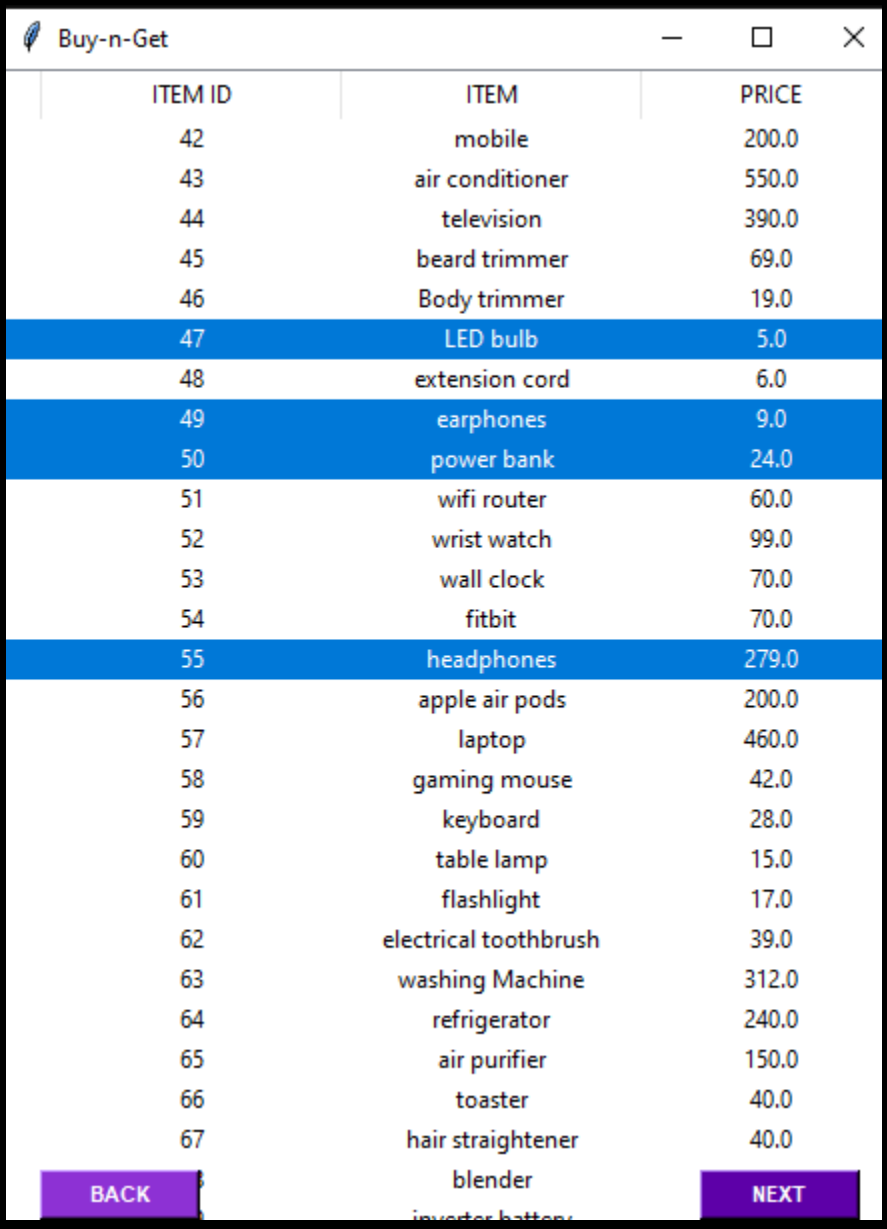

Buy-n-Get
Back
Buy-n-Get
Login
Username
vineet
Password
123qwe123
Done
Logged In
Welcome vineet !
OK

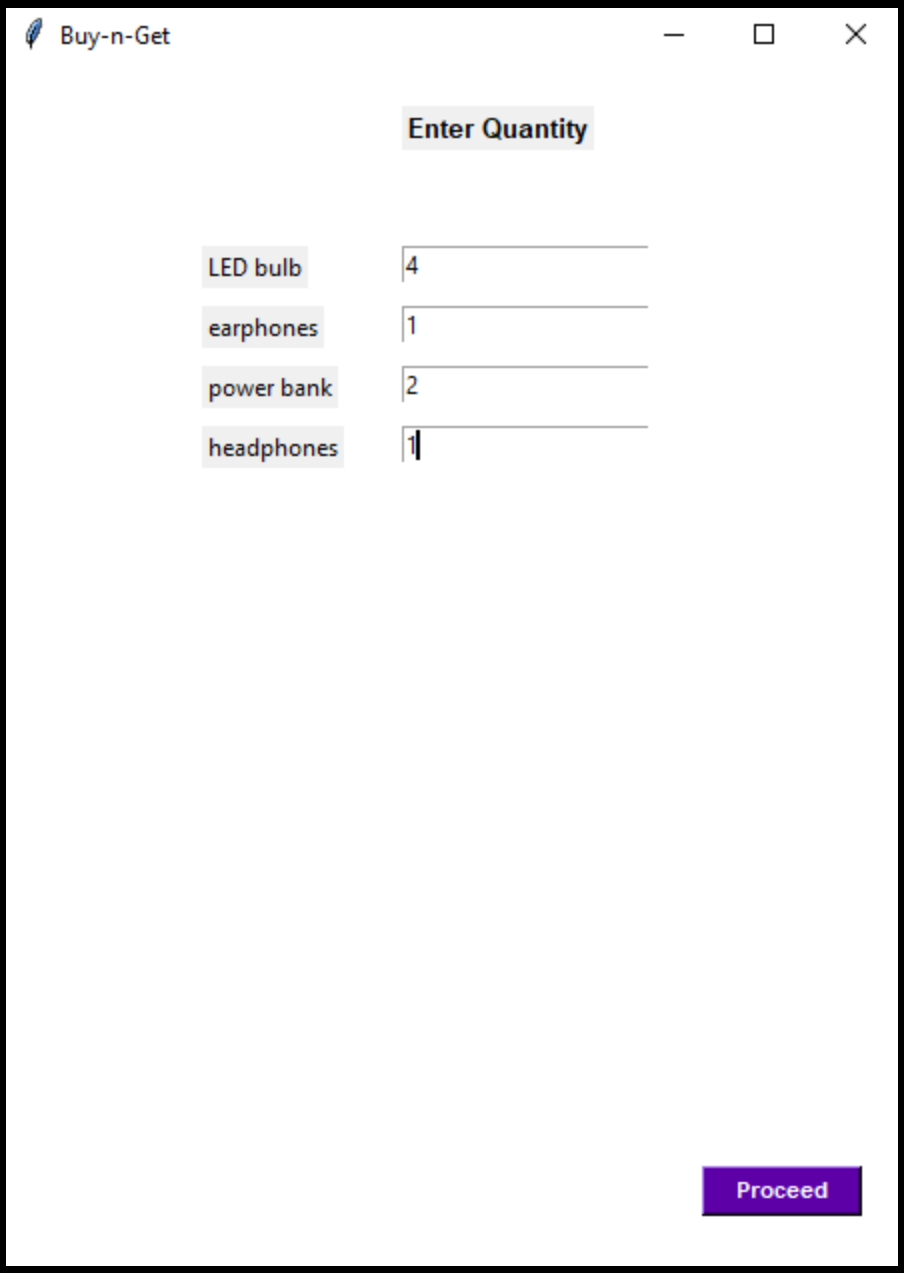

Buy-n-Get
Back
Sign in as...
BUYER
SELLER

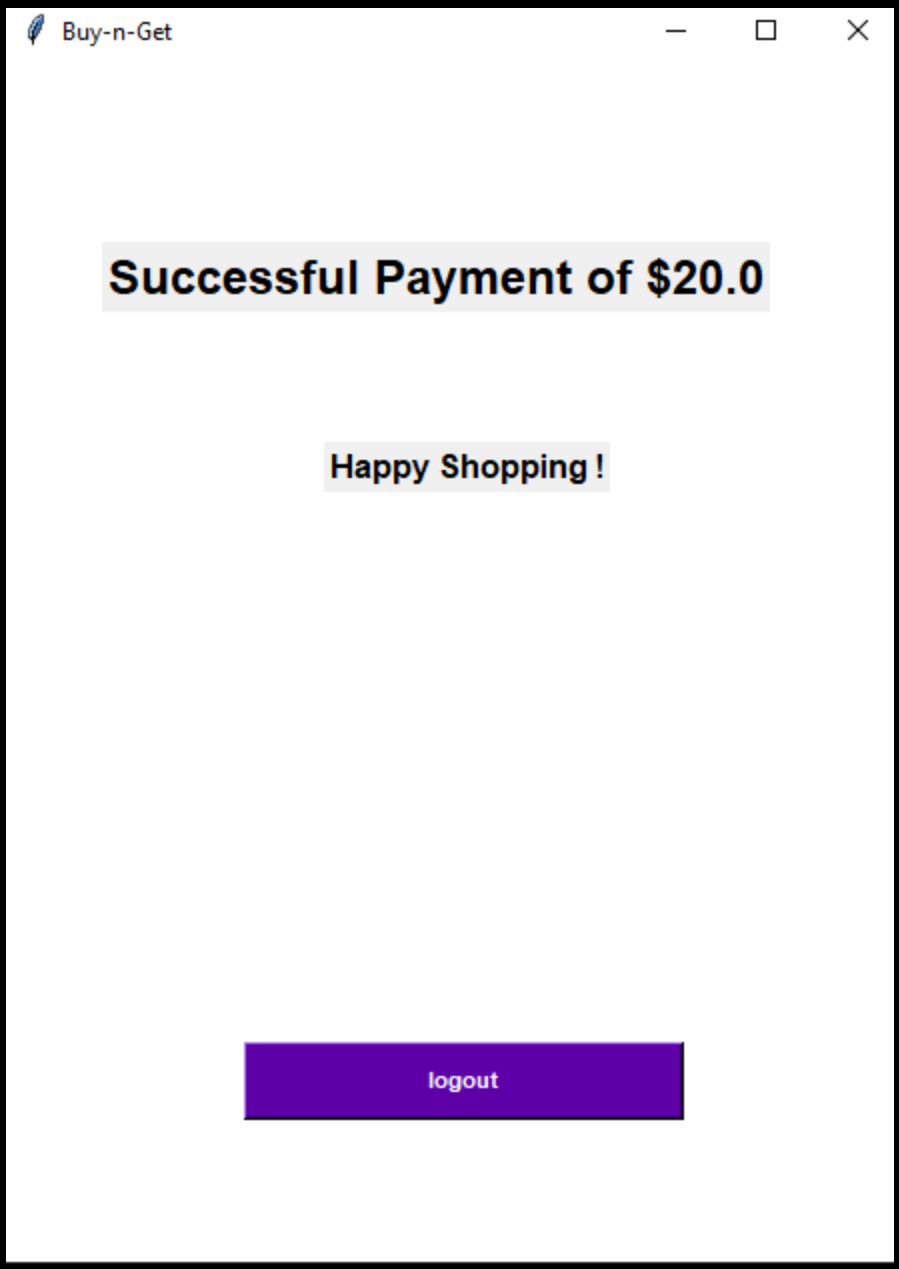
Buyer Walkthrough



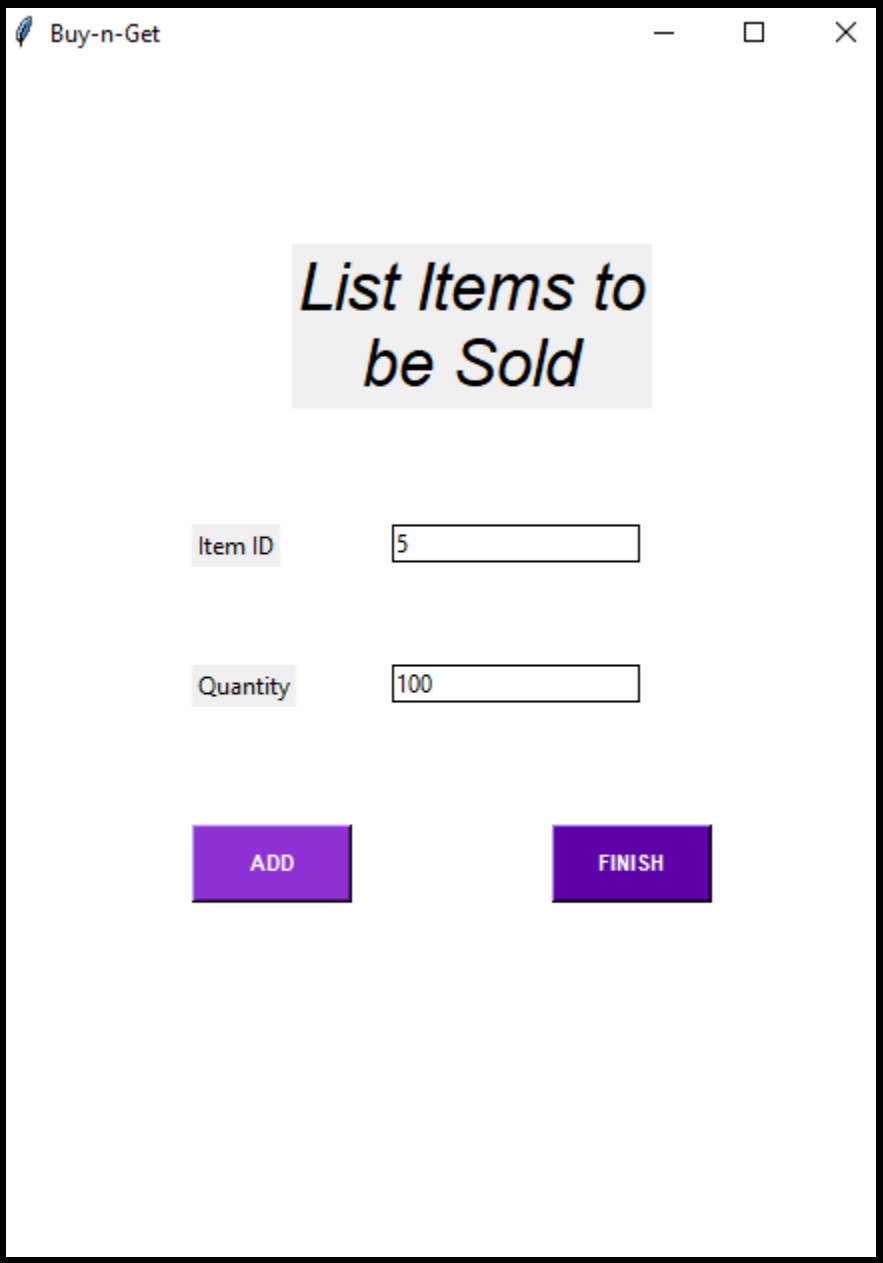
(Inside Electronics)

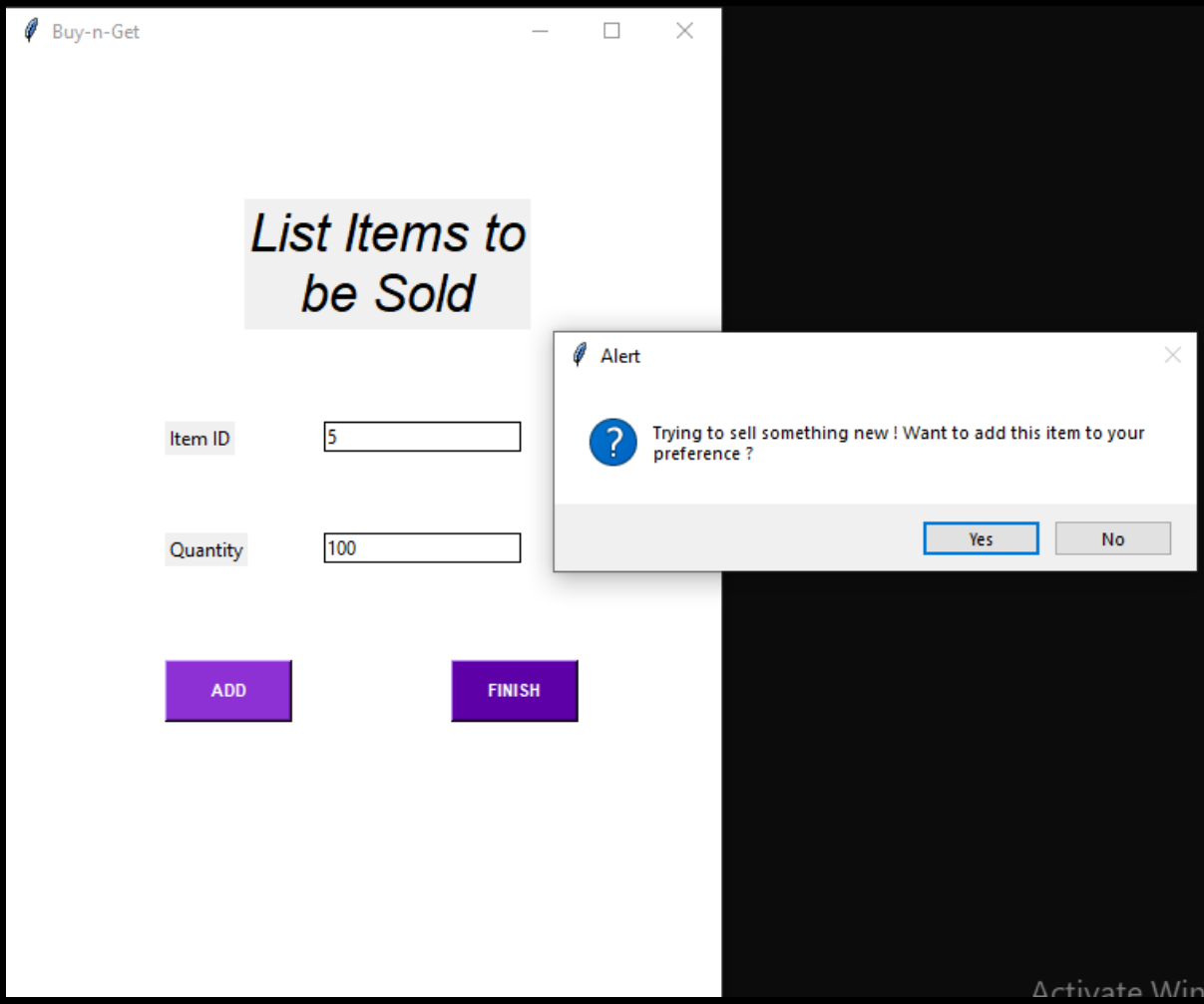


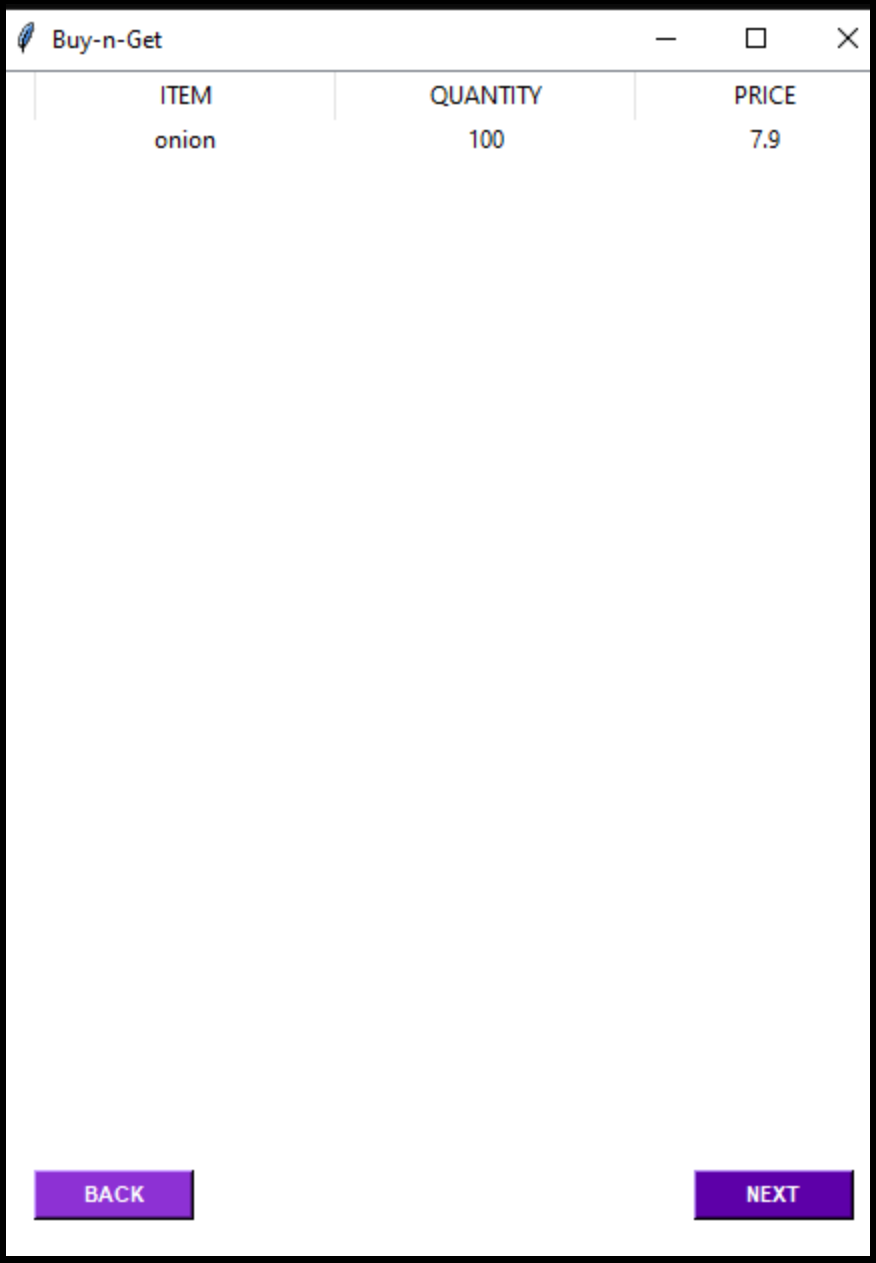

Buy-n-Get
Enter Quantity
LED bulb
4
earphones
1
power bank
2
headphones
1
Proceed

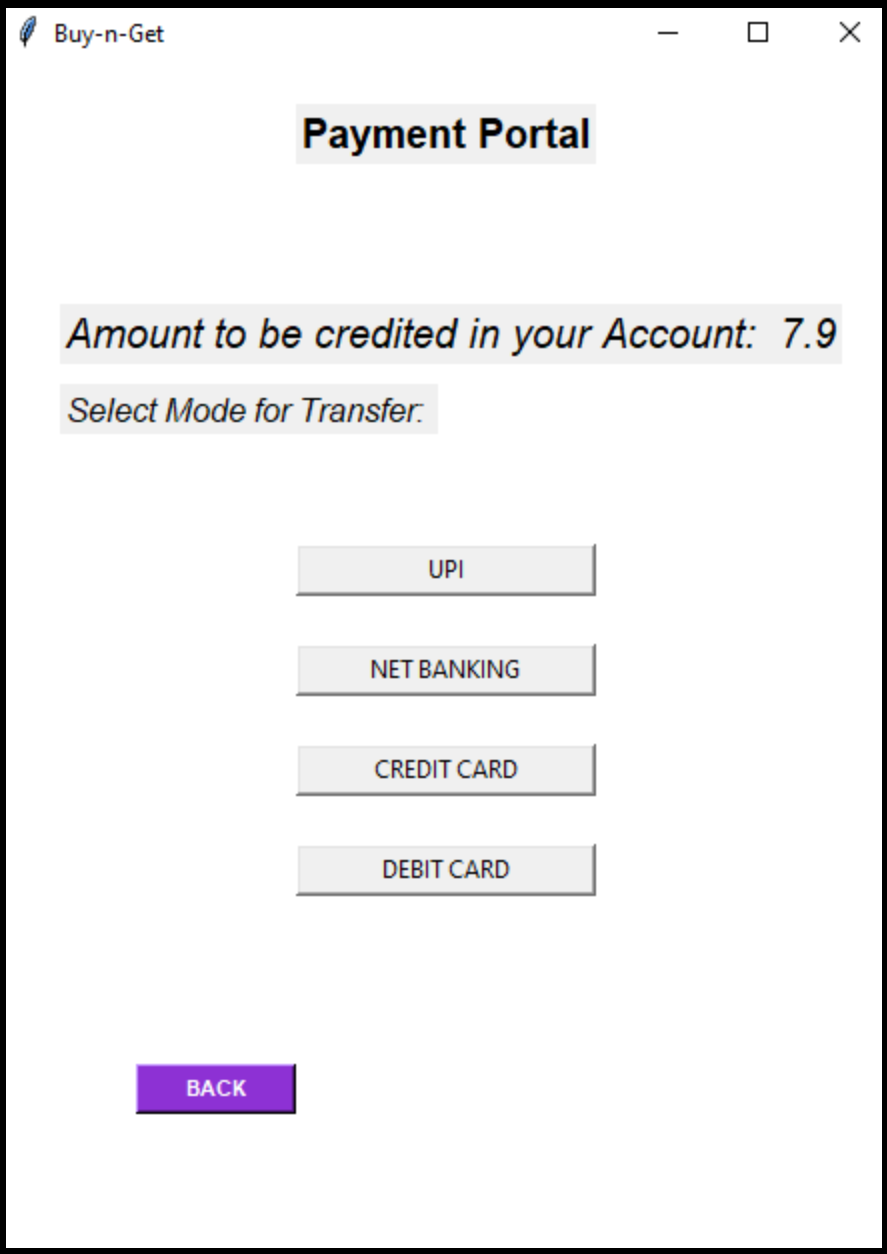

Buy-n-Get
Successful Payment of $20.0
Happy Shopping !
logout

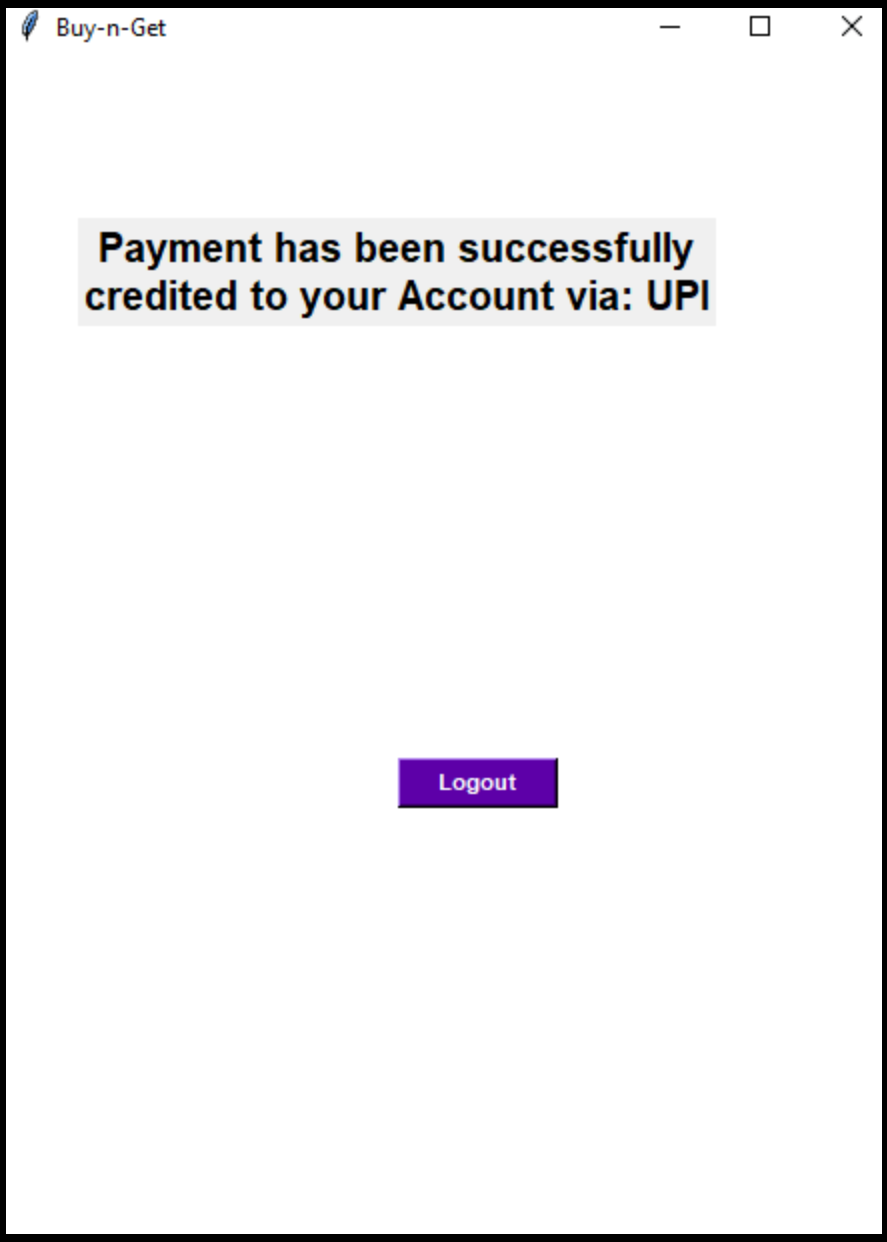
Seller Walkthrough




Buy-n-Get
List Items to be Sold
Item ID
5
Quantity
100
ADD
FINISH
Alert
Trying to sell something new ! Want to add this item to your preference ?
Yes
No


Buy-n-Get
ITEM
QUANTITY
PRICE
onion
100
7.9
BACK
NEXT


Buy-n-Get
Payment Portal
Amount to be credited in your Account: 7.9
Select Mode for Transfer:
UPI
NET BANKING
CREDIT CARD
DEBIT CARD
BACK


Buy-n-Get
Payment has been successfully credited to your Account via: UPI
Logout

USP/Added functionality in the project

1. We have developed a fully-functioning user interface for our database.
2. Implemented some key features like real time Signing-Up and Login services.
3. Special attention paid on the aesthetics, and to make a user-friendly and easy-to-learn Graphical User Interface.
4. Cart functionality is provided and maintained at both database and GUI paradigms.
5. Submitted quite complex and extremely optimized SQL queries.