Criteria C – Product Development

Techniques Used

1. **Database implementation and normalization:** Normalization, validation, constraints, and checks used.

2. **Classes and Objects:** Classes, objects, and methods used to manage website's database using Python.

3. **Updating Data:** JavaScript event listeners, HTTP requests, and functions used.

4. **Managing Checkout Functionality & Integrating APIs:** Functions, HTTP requests, and APIs used.

5. **Functions Used:** To implement a DRY methodology.

6. **User Interface:** login validations, interface modifications, alerts, frontend-exception-handling, and minimal text entry used to enhance the website's usability.

7. **Searching and Filtering:** Allowing searching and filtering basis specified parameters for both customers and administrators.

8. **Access Rights and Data Security:** Managing administrator access rights and data storage formats.

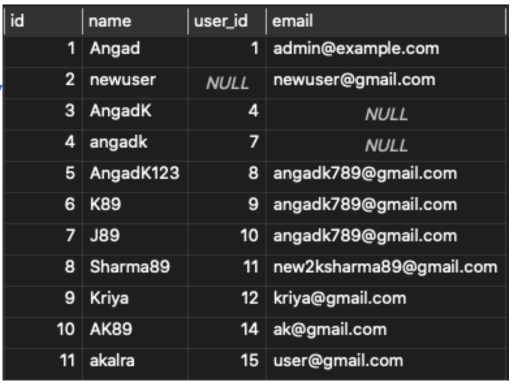
9. **Nested Loops:** Nested loops used to output details of previously-placed orders.

Database Implementation

1. **Database normalized in 3rd Normal Form to remove partial and transitive functional dependencies.**

All tables are in 1NF as all fields store atomic values and each table has a unique, auto-incrementing primary key identifier.

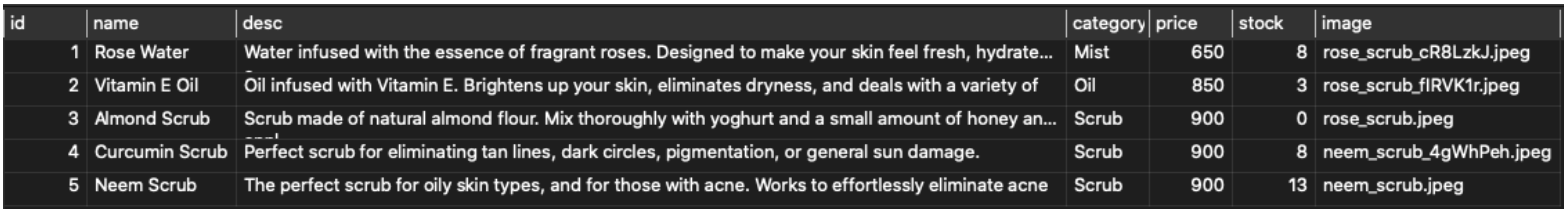
Customer Table



id	name	user_id	email
1	Angad	1	admin@example.com
2	newuser	NULL	newuser@gmail.com
3	AngadK	4	NULL
4	angadk	7	NULL
5	AngadK123	8	angadk789@gmail.com
6	K89	9	angadk789@gmail.com
7	J89	10	angadk789@gmail.com
8	Sharma89	11	new2ksharma89@gmail.com
9	Kriya	12	kriya@gmail.com
10	AK89	14	ak@gmail.com
11	akalra	15	user@gmail.com

Auto-incrementing primary keys

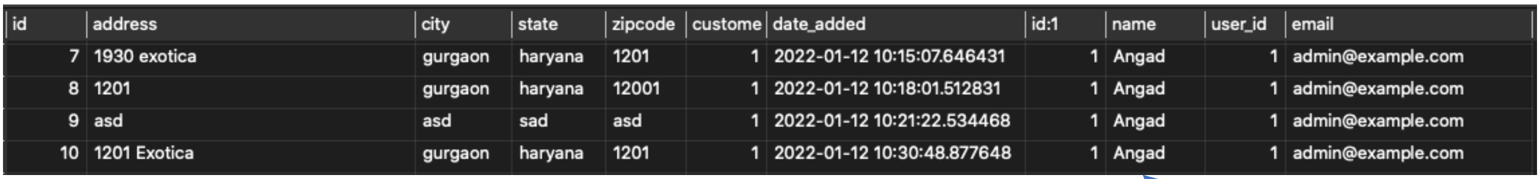
Products Table



id	name	desc	category	price	stock	image
1	Rose Water	Water infused with the essence of fragrant roses. Designed to make your skin feel fresh, hydrate...	Mist	650	8	rose_scrub_cR8LzkJ.jpeg
2	Vitamin E Oil	Oil infused with Vitamin E. Brightens up your skin, eliminates dryness, and deals with a variety of	Oil	850	3	rose_scrub_fIRVK1r.jpeg
3	Almond Scrub	Scrub made of natural almond flour. Mix thoroughly with yoghurt and a small amount of honey an...	Scrub	900	0	rose_scrub.jpeg
4	Curcumin Scrub	Perfect scrub for eliminating tan lines, dark circles, pigmentation, or general sun damage.	Scrub	900	8	neem_scrub_4gWhPeh.jpeg
5	Neem Scrub	The perfect scrub for oily skin types, and for those with acne. Works to effortlessly eliminate acne	Scrub	900	13	neem_scrub.jpeg

The tables were already in 2NF as they had no partial dependencies. However, certain transitive dependencies were removed to place the table in 3NF and minimize data redundancy:

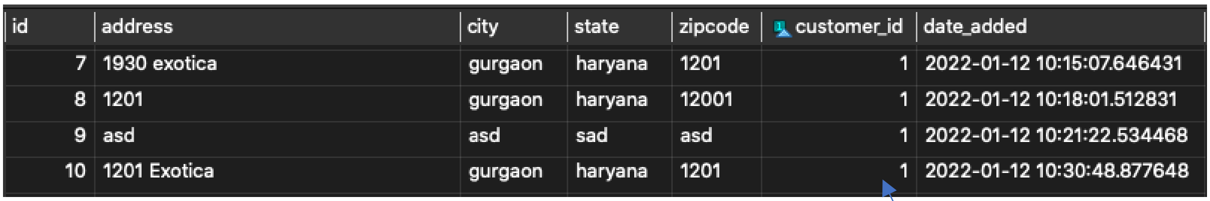
Shipping Addresses Table (un-normalized)



id	address	city	state	zipcode	custome	date_added	id:1	name	user_id	email
7	1930 exotica	gurgaon	haryana	1201	1	2022-01-12 10:15:07.646431	1	Angad	1	admin@example.com
8	1201	gurgaon	haryana	12001	1	2022-01-12 10:18:01.512831	1	Angad	1	admin@example.com
9	asd	asd	sad	asd	1	2022-01-12 10:21:22.534468	1	Angad	1	admin@example.com
10	1201 Exotica	gurgaon	haryana	1201	1	2022-01-12 10:30:48.877648	1	Angad	1	admin@example.com

All customer data transitively dependent on 'id:1' field

Shipping Addresses Table (3NF)



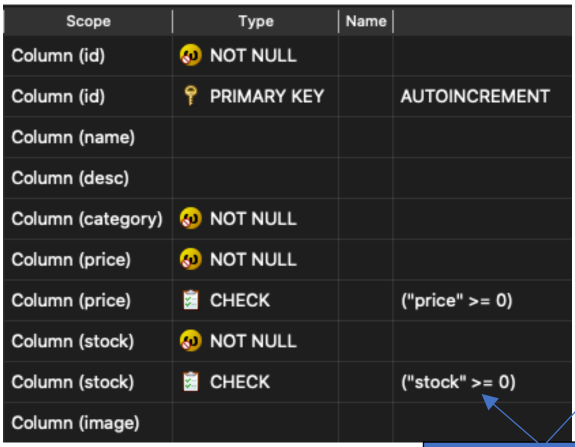
id	address	city	state	zipcode	customer_id	date_added
7	1930 exotica	gurgaon	haryana	1201	1	2022-01-12 10:15:07.646431
8	1201	gurgaon	haryana	12001	1	2022-01-12 10:18:01.512831
9	asd	asd	sad	asd	1	2022-01-12 10:21:22.534468
10	1201 Exotica	gurgaon	haryana	1201	1	2022-01-12 10:30:48.877648

Functionality preserved – customers can store multiple addresses

Created new table for customer data + inserted referential key

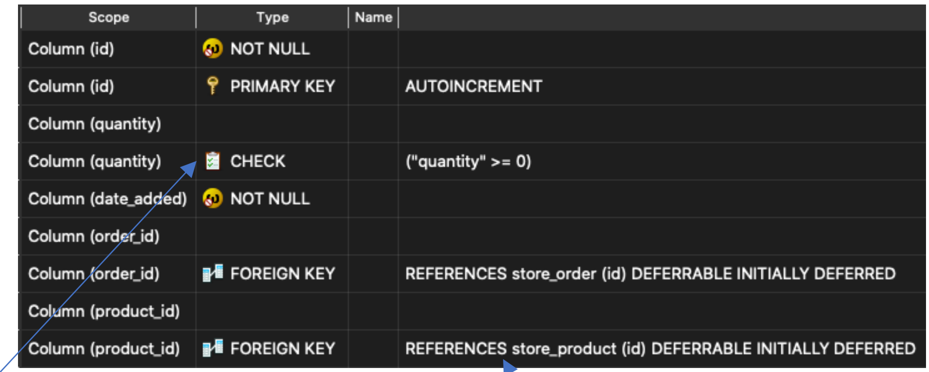
2. Data constraints, validations, and checks used to ensure accuracy of stored data.

Products Table



Scope	Type	Name	
Column (id)	NOT NULL		
Column (id)	PRIMARY KEY		AUTOINCREMENT
Column (name)			
Column (desc)			
Column (category)	NOT NULL		
Column (price)	NOT NULL		
Column (price)	CHECK		("price" >= 0)
Column (stock)	NOT NULL		
Column (stock)	CHECK		("stock" >= 0)
Column (image)			

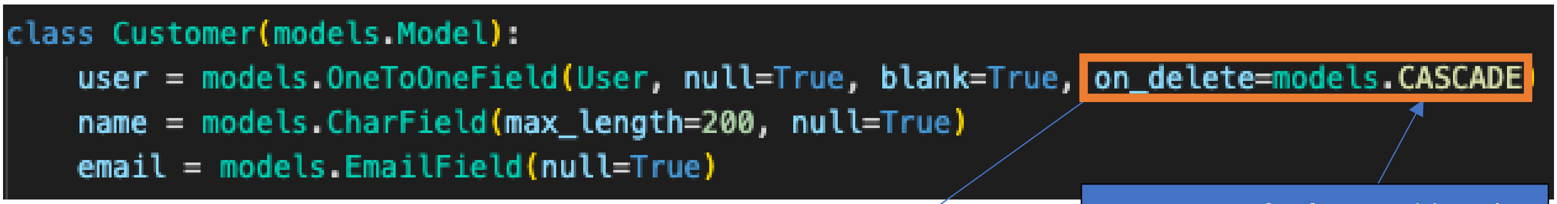
OrderItem Table



Scope	Type	Name	
Column (id)	NOT NULL		
Column (id)	PRIMARY KEY		AUTOINCREMENT
Column (quantity)			
Column (quantity)	CHECK		("quantity" >= 0)
Column (date_added)	NOT NULL		
Column (order_id)			
Column (order_id)	FOREIGN KEY		REFERENCES store_order (id) DEFERRABLE INITIALLY DEFERRED
Column (product_id)			
Column (product_id)	FOREIGN KEY		REFERENCES store_product (id) DEFERRABLE INITIALLY DEFERRED

Check ensures price/stock/quantity are always positive (≥ 0)

Referential integrity constraint placed on foreign keys



```
class Customer(models.Model):
    user = models.OneToOneField(User, null=True, blank=True, on_delete=models.CASCADE)
    name = models.CharField(max_length=200, null=True)
    email = models.EmailField(null=True)
```

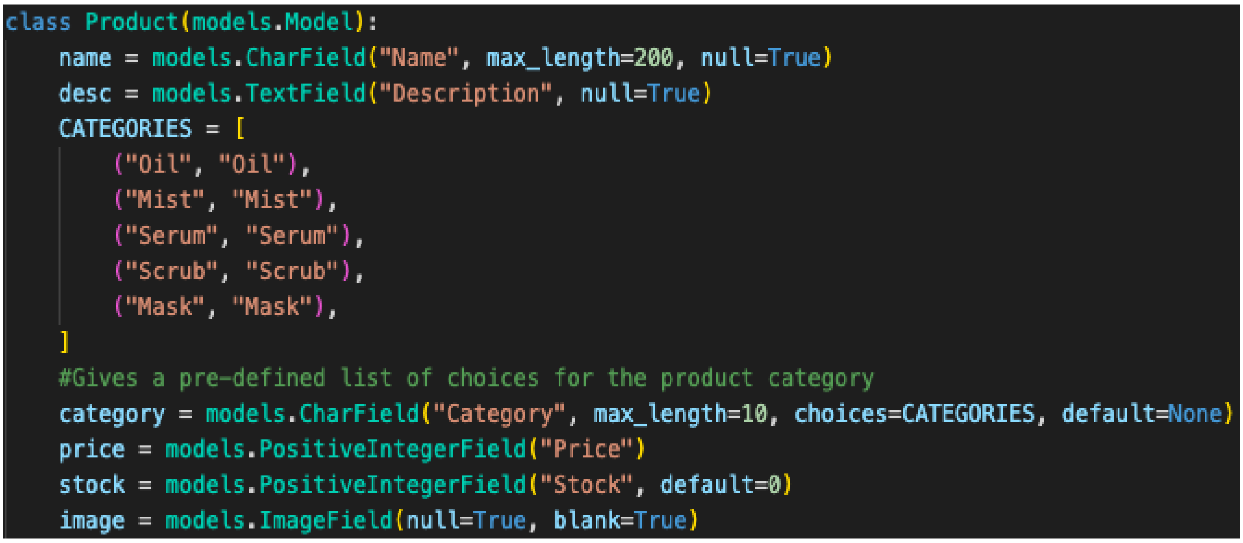
On_delete parameter manages referential integrity constraint

CASCADE – if referenced ‘user’ record deleted, FK ‘customer’ record also deleted – prevents data inconsistency

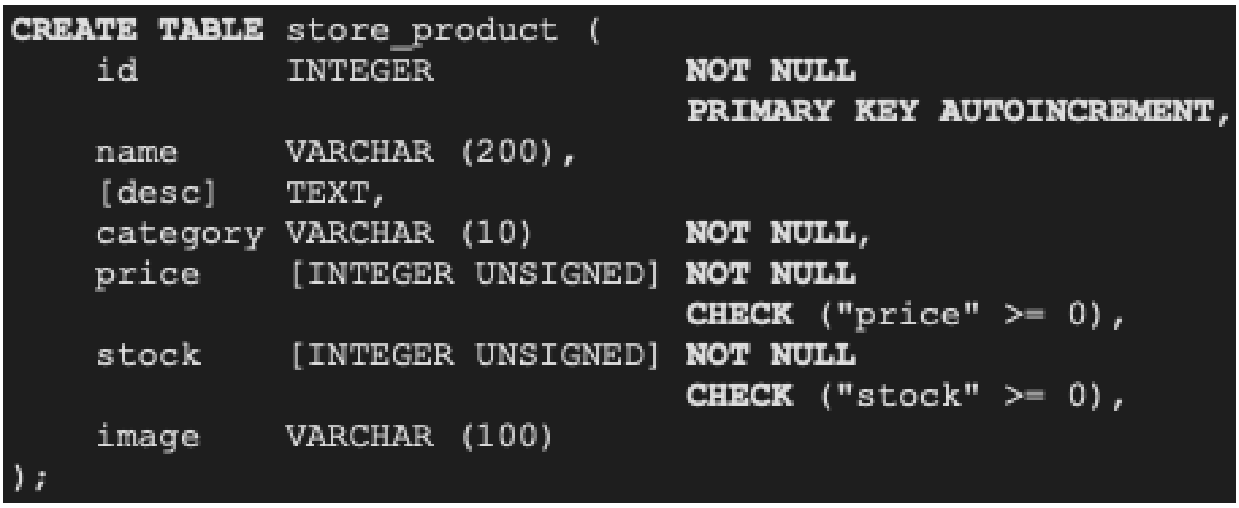
Classes and Objects

Each database table is defined as a Python class which inherit from Django’s base ‘models.Model’ class (converts Python commands into SQL/database-readable commands).

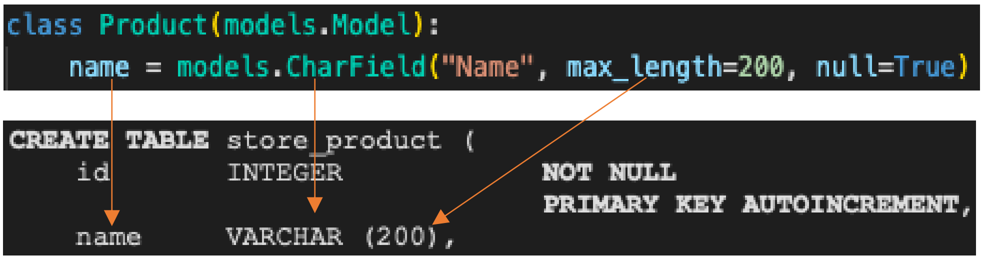
E.g., Comparing the SQL DDL and Python class for the Products table:



```
class Product(models.Model):
    name = models.CharField("Name", max_length=200, null=True)
    desc = models.TextField("Description", null=True)
    CATEGORIES = [
        ("Oil", "Oil"),
        ("Mist", "Mist"),
        ("Serum", "Serum"),
        ("Scrub", "Scrub"),
        ("Mask", "Mask"),
    ]
    #Gives a pre-defined list of choices for the product category
    category = models.CharField("Category", max_length=10, choices=CATEGORIES, default=None)
    price = models.PositiveIntegerField("Price")
    stock = models.PositiveIntegerField("Stock", default=0)
    image = models.ImageField(null=True, blank=True)
```



```sql
CREATE TABLE store_product (
    id       INTEGER              NOT NULL
                                  PRIMARY KEY AUTOINCREMENT,
    name     VARCHAR (200),
    [desc]   TEXT,
    category VARCHAR (10)         NOT NULL,
    price    [INTEGER UNSIGNED]   NOT NULL
                                  CHECK ("price" >= 0),
    stock    [INTEGER UNSIGNED]   NOT NULL
                                  CHECK ("stock" >= 0),
    image    VARCHAR (100)
);
```



```python
class Product(models.Model):
    name = models.CharField("Name", max_length=200, null=True)
```

```sql
CREATE TABLE store_product (
    id       INTEGER              NOT NULL
                                  PRIMARY KEY AUTOINCREMENT,
    name     VARCHAR (200),
```

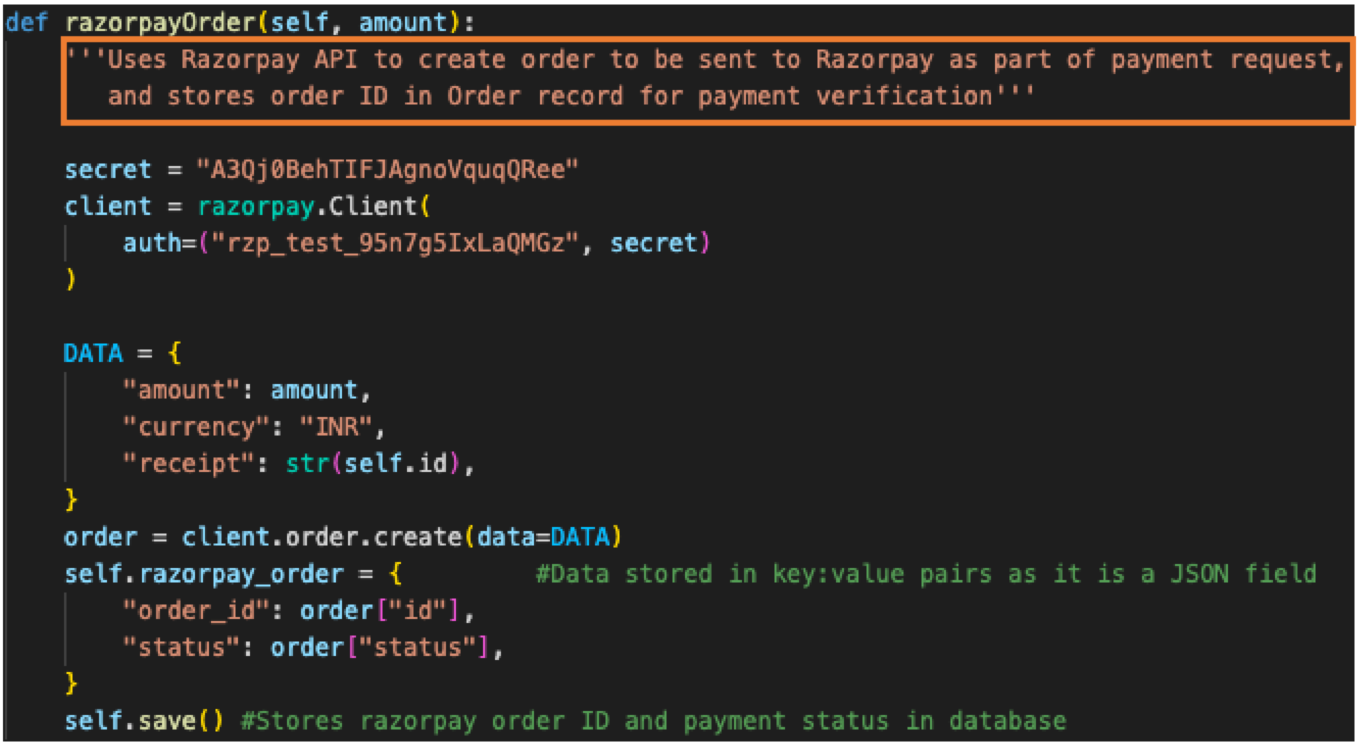
Database table field → Class attribute

Each class attribute is an instance of a Django predefined class, basis the data type of the database field.

Field constraints → Attributes of instance

These classes also utilize **public methods and getters.**

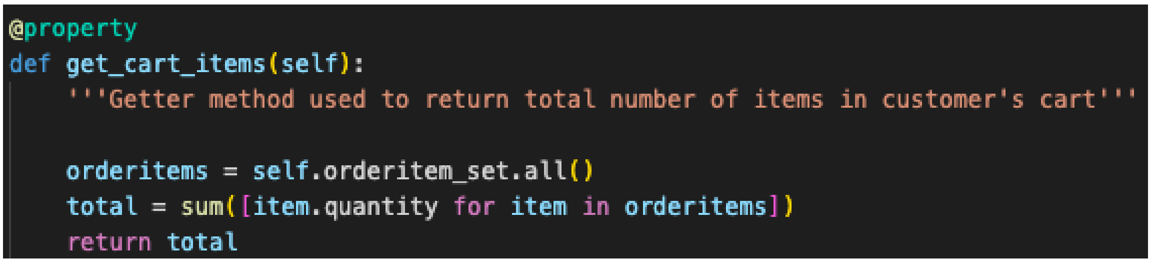
E.g., **razorpayOrder:** Public method used when paying via Razorpay.



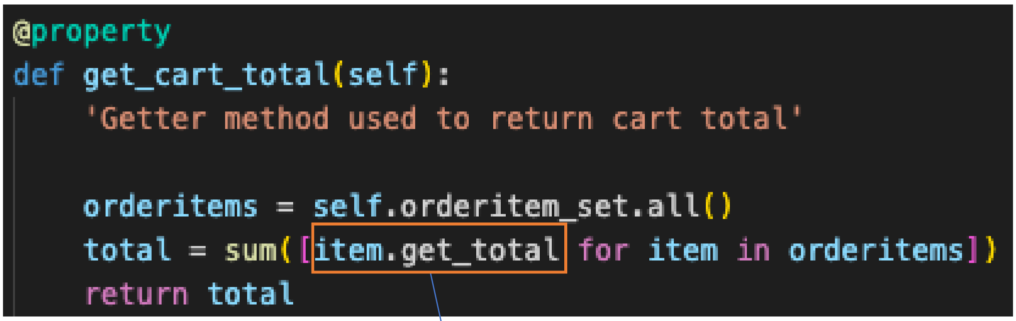
```python
def razorpayOrder(self, amount):
    '''Uses Razorpay API to create order to be sent to Razorpay as part of payment request,
    and stores order ID in Order record for payment verification'''

    secret = "A3Qj0BehTIFJAgnoVquqQRee"
    client = razorpay.Client(
        auth=("rzp_test_95n7g5IxLaQMGz", secret)
    )

    DATA = {
        "amount": amount,
        "currency": "INR",
        "receipt": str(self.id),
    }
    order = client.order.create(data=DATA)
    self.razorpay_order = {          #Data stored in key:value pairs as it is a JSON field
        "order_id": order["id"],
        "status": order["status"],
    }
    self.save() #Stores razorpay order ID and payment status in database
```

@property decorator creates getters – Methods can only query/operate on data and return an output to user. They **cannot update the object's attributes.**

Order class getters:



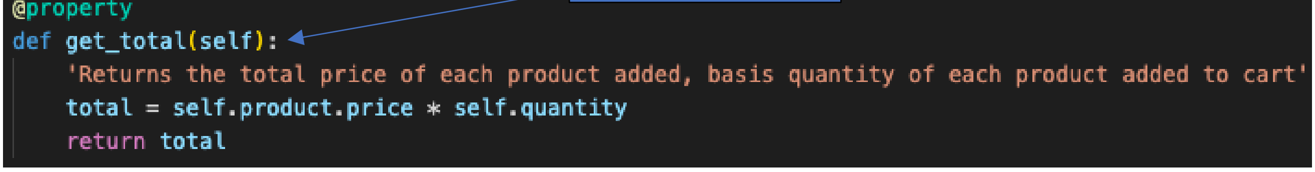
```python
@property
def get_cart_items(self):
    '''Getter method used to return total number of items in customer's cart'''

    orderitems = self.orderitem_set.all()
    total = sum([item.quantity for item in orderitems])
    return total
```



```python
@property
def get_cart_total(self):
    'Getter method used to return cart total'

    orderitems = self.orderitem_set.all()
    total = sum([item.get_total for item in orderitems])
    return total
```

References OrderItem getter

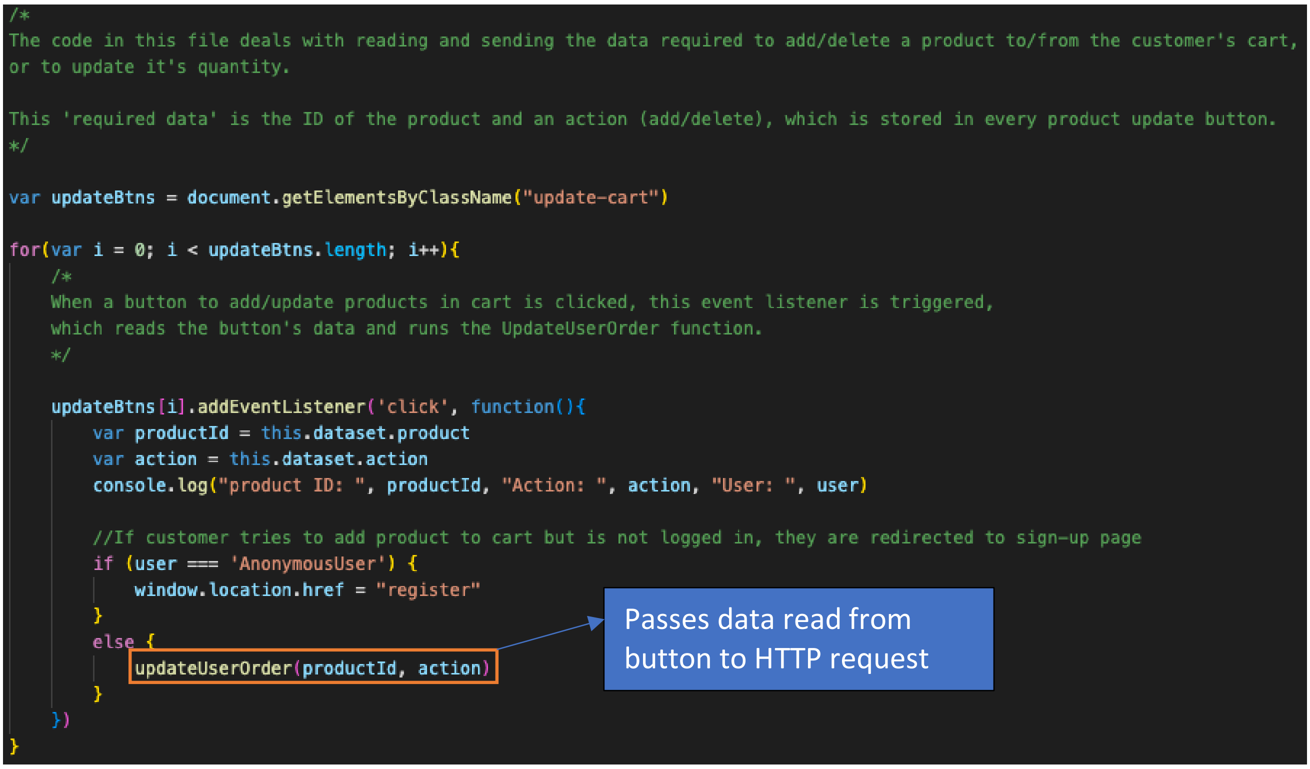
OrderItem class getters:



```python
@property
def get_total(self):
    'Returns the total price of each product added, basis quantity of each product added to cart'
    total = self.product.price * self.quantity
    return total
```

Updating Data

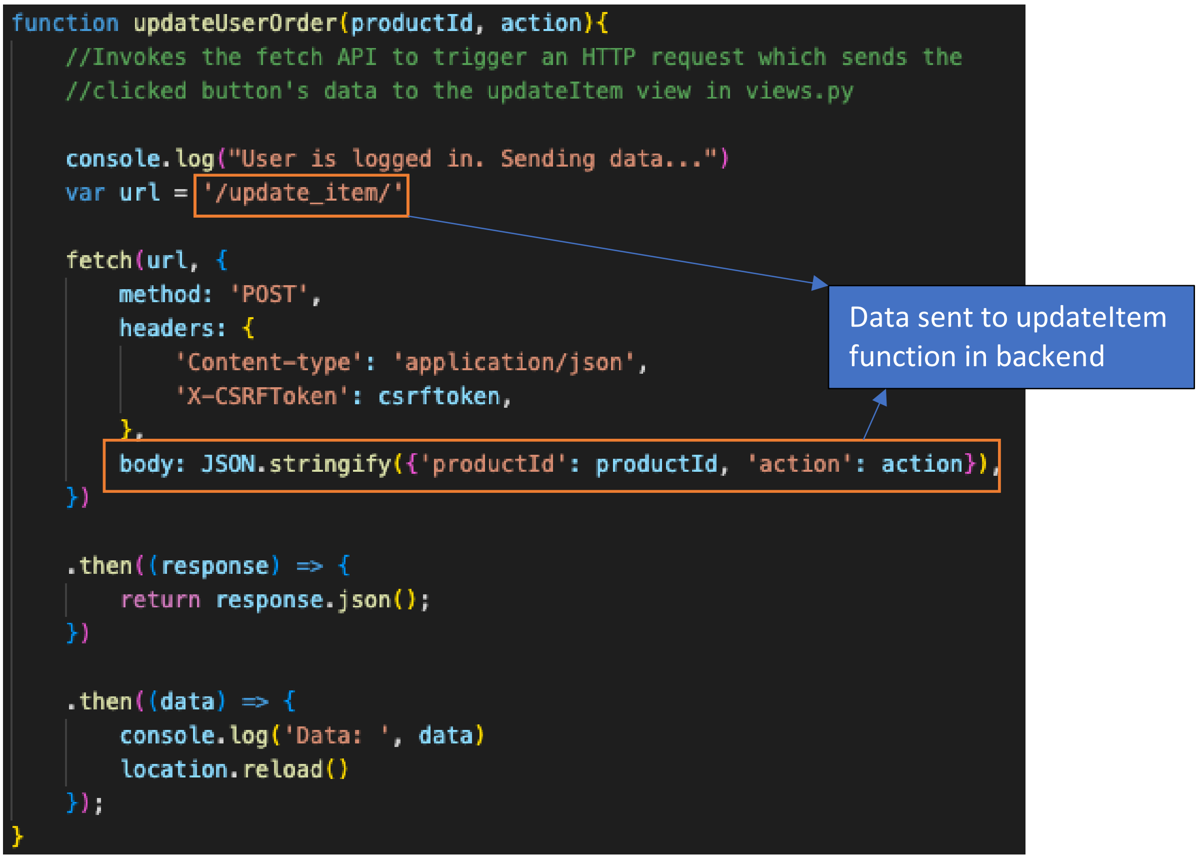
EventListener listens for and reads user input, and then triggers relevant HTTP request to send input to backend for processing. E.g., the process of adding/removing a product to/from the cart:



```javascript
/*
The code in this file deals with reading and sending the data required to add/delete a product to/from the customer's cart,
or to update it's quantity.

This 'required data' is the ID of the product and an action (add/delete), which is stored in every product update button.
*/

var updateBtns = document.getElementsByClassName("update-cart")

for(var i = 0; i < updateBtns.length; i++){
    /*
    When a button to add/update products in cart is clicked, this event listener is triggered,
    which reads the button's data and runs the UpdateUserOrder function.
    */

    updateBtns[i].addEventListener('click', function(){
        var productId = this.dataset.product
        var action = this.dataset.action
        console.log("product ID: ", productId, "Action: ", action, "User: ", user)

        //If customer tries to add product to cart but is not logged in, they are redirected to sign-up page
        if (user === 'AnonymousUser') {
            window.location.href = "register"
        }
        else {
            updateUserOrder(productId, action)
        }
    })
}
```

Passes data read from button to HTTP request

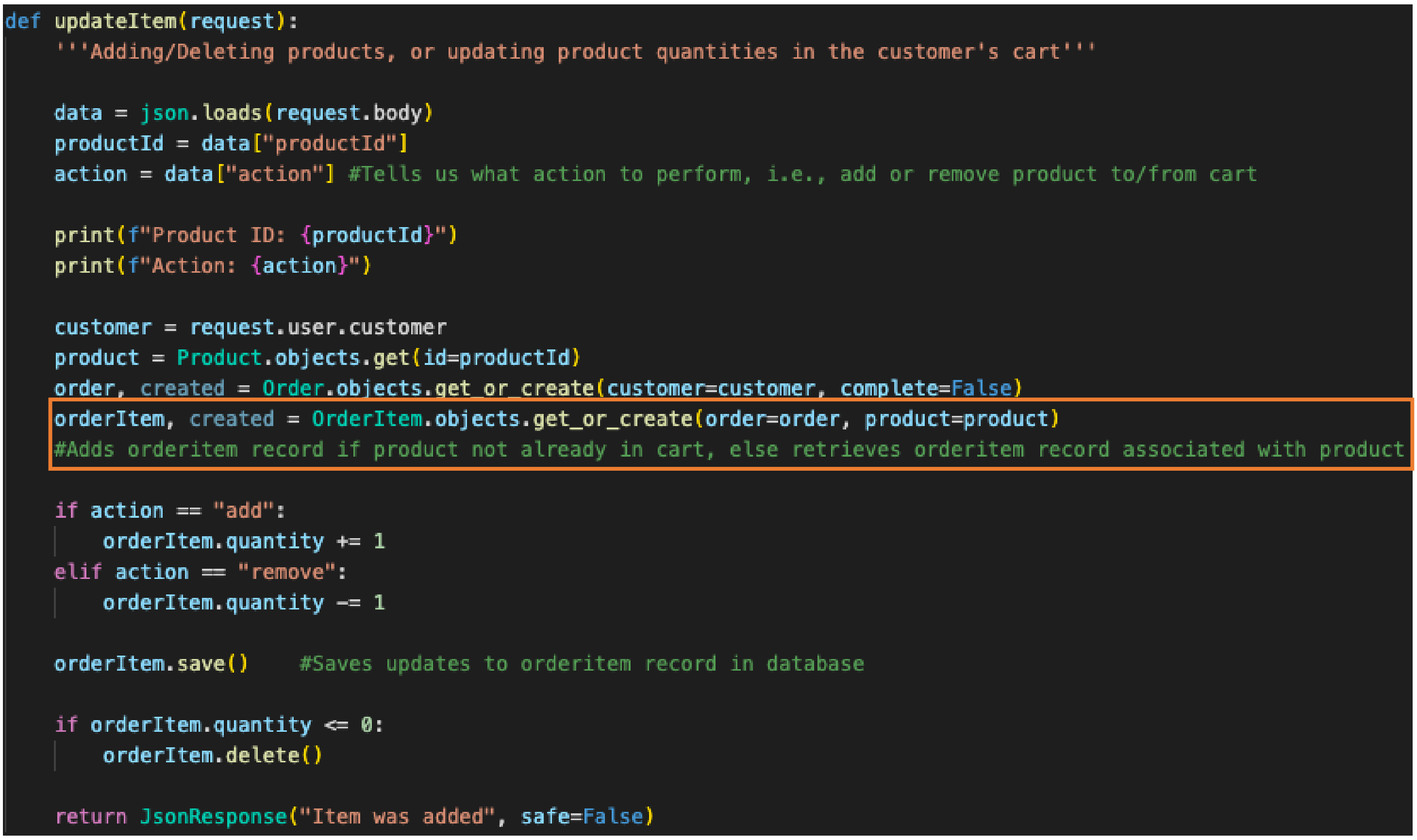
A JavaScript Fetch API call then sends the HTTP POST request to backend for processing.



```
function updateUserOrder(productId, action){
    //Invokes the fetch API to trigger an HTTP request which sends the
    //clicked button's data to the updateItem view in views.py

    console.log("User is logged in. Sending data...")
    var url = '/update_item/'

    fetch(url, {
        method: 'POST',
        headers: {
            'Content-type': 'application/json',
            'X-CSRFToken': csrftoken,
        },
        body: JSON.stringify({'productId': productId, 'action': action}),
    })

    .then((response) => {
        return response.json();
    })

    .then((data) => {
        console.log('Data: ', data)
        location.reload()
    });
}
```

Data sent to updateItem function in backend

Data sent by request to add/update/delete products is then updated and committed to database by updateItem():

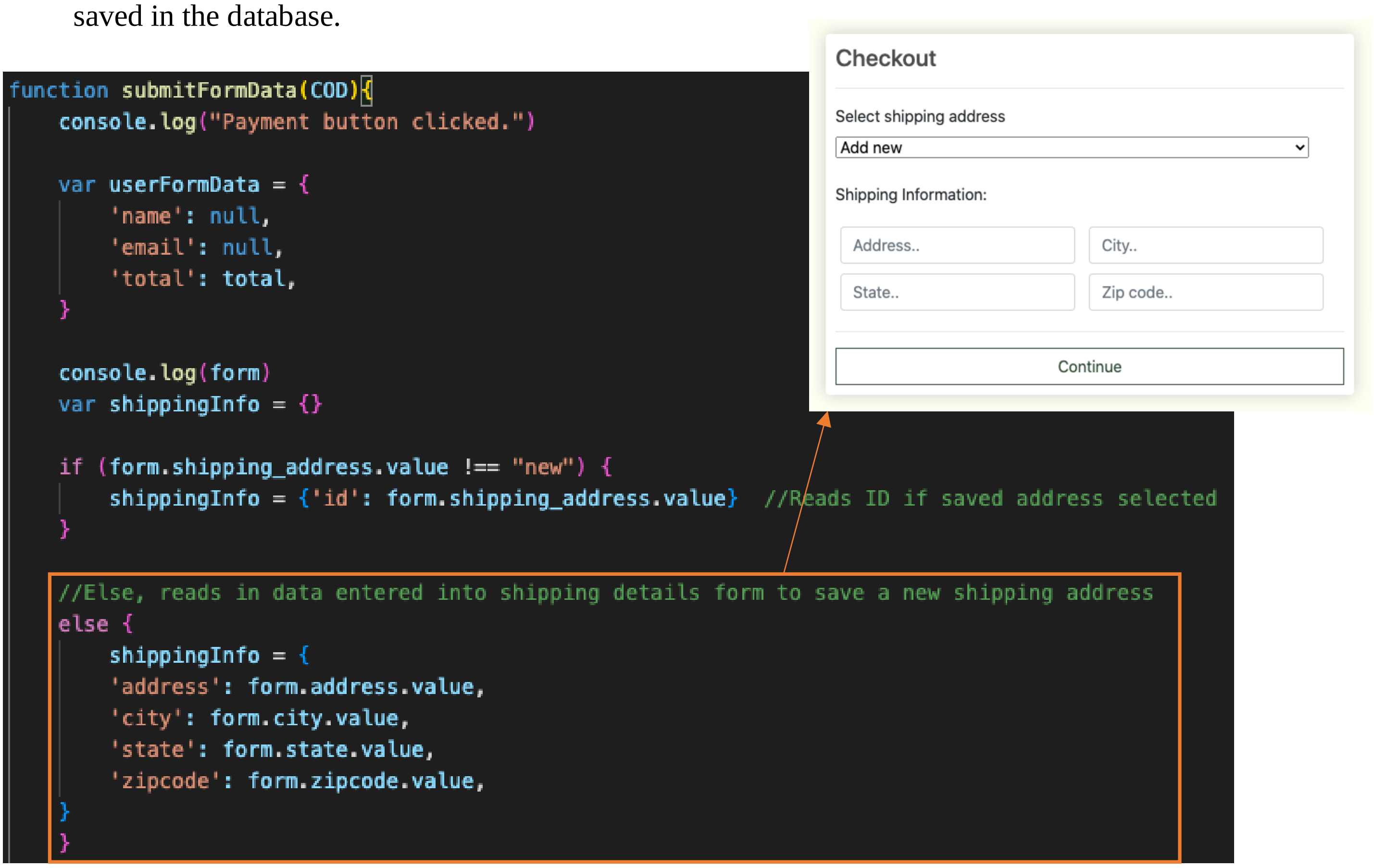


```
def updateItem(request):
    '''Adding/Deleting products, or updating product quantities in the customer's cart'''

    data = json.loads(request.body)
    productId = data["productId"]
    action = data["action"] #Tells us what action to perform, i.e., add or remove product to/from cart

    print(f"Product ID: {productId}")
    print(f"Action: {action}")

    customer = request.user.customer
    product = Product.objects.get(id=productId)
    order, created = Order.objects.get_or_create(customer=customer, complete=False)
    orderItem, created = OrderItem.objects.get_or_create(order=order, product=product)
    #Adds orderitem record if product not already in cart, else retrieves orderitem record associated with product

    if action == "add":
        orderItem.quantity += 1
    elif action == "remove":
        orderItem.quantity -= 1

    orderItem.save()    #Saves updates to orderitem record in database

    if orderItem.quantity <= 0:
        orderItem.delete()

    return JsonResponse("Item was added", safe=False)
```

Managing Checkout Functionality & Integrating APIs

Use of multiple functions and Fetch API calls (HTTP requests) to fulfil checkout functionality, reduce errors, and increase efficiency. External Fetch & Razorpay APIs were also integrated in the process.

First, the customer's shipping details are read and sent to backend via a Fetch API call, to be saved in the database.



```
function submitFormData(COD){
    console.log("Payment button clicked.")

    var userFormData = {
        'name': null,
        'email': null,
        'total': total,
    }

    console.log(form)
    var shippingInfo = {}

    if (form.shipping_address.value !== "new") {
        shippingInfo = {'id': form.shipping_address.value}  //Reads ID if saved address selected
    }

    //Else, reads in data entered into shipping details form to save a new shipping address
    else {
        shippingInfo = {
        'address': form.address.value,
        'city': form.city.value,
        'state': form.state.value,
        'zipcode': form.zipcode.value,
    }
    }
```

Checkout

Select shipping address

Add new

Shipping Information:

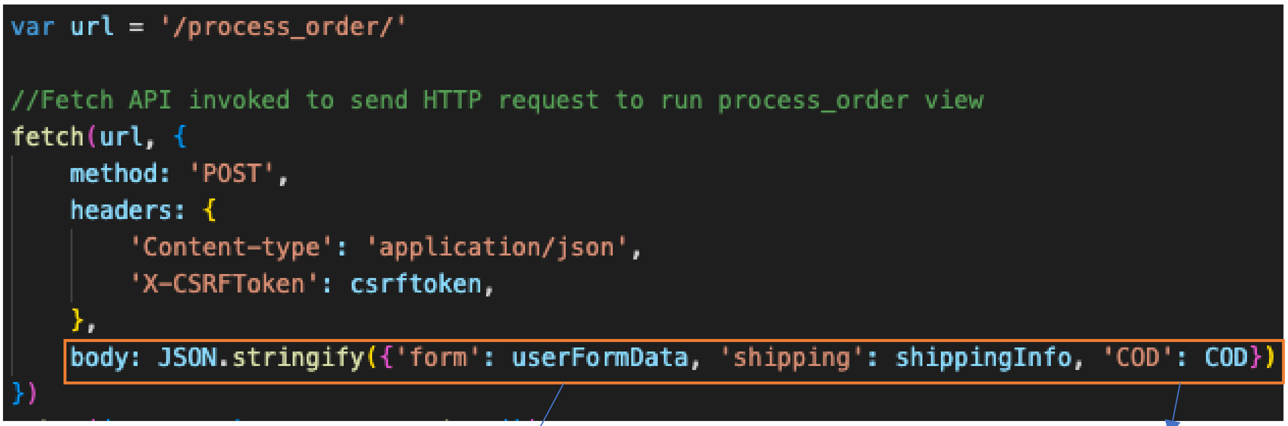
Address..

City..

State..

Zip code..

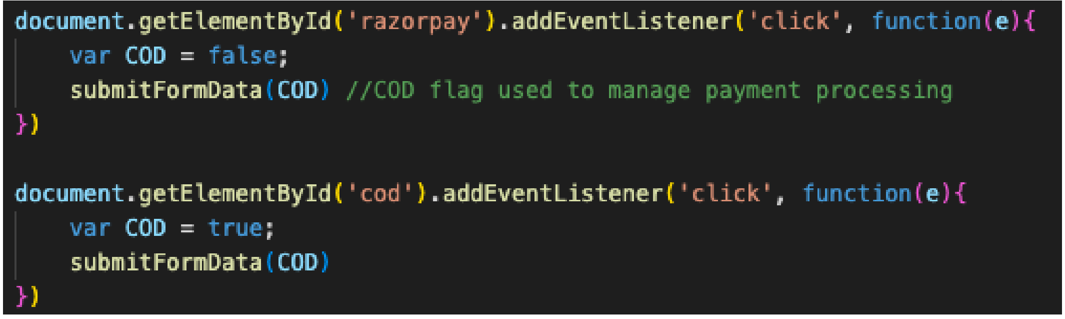
Continue



```
var url = '/process_order/'

//Fetch API invoked to send HTTP request to run process_order view
fetch(url, {
    method: 'POST',
    headers: {
        'Content-type': 'application/json',
        'X-CSRFToken': csrftoken,
    },
    body: JSON.stringify({'form': userFormData, 'shipping': shippingInfo, 'COD': COD})
})
```

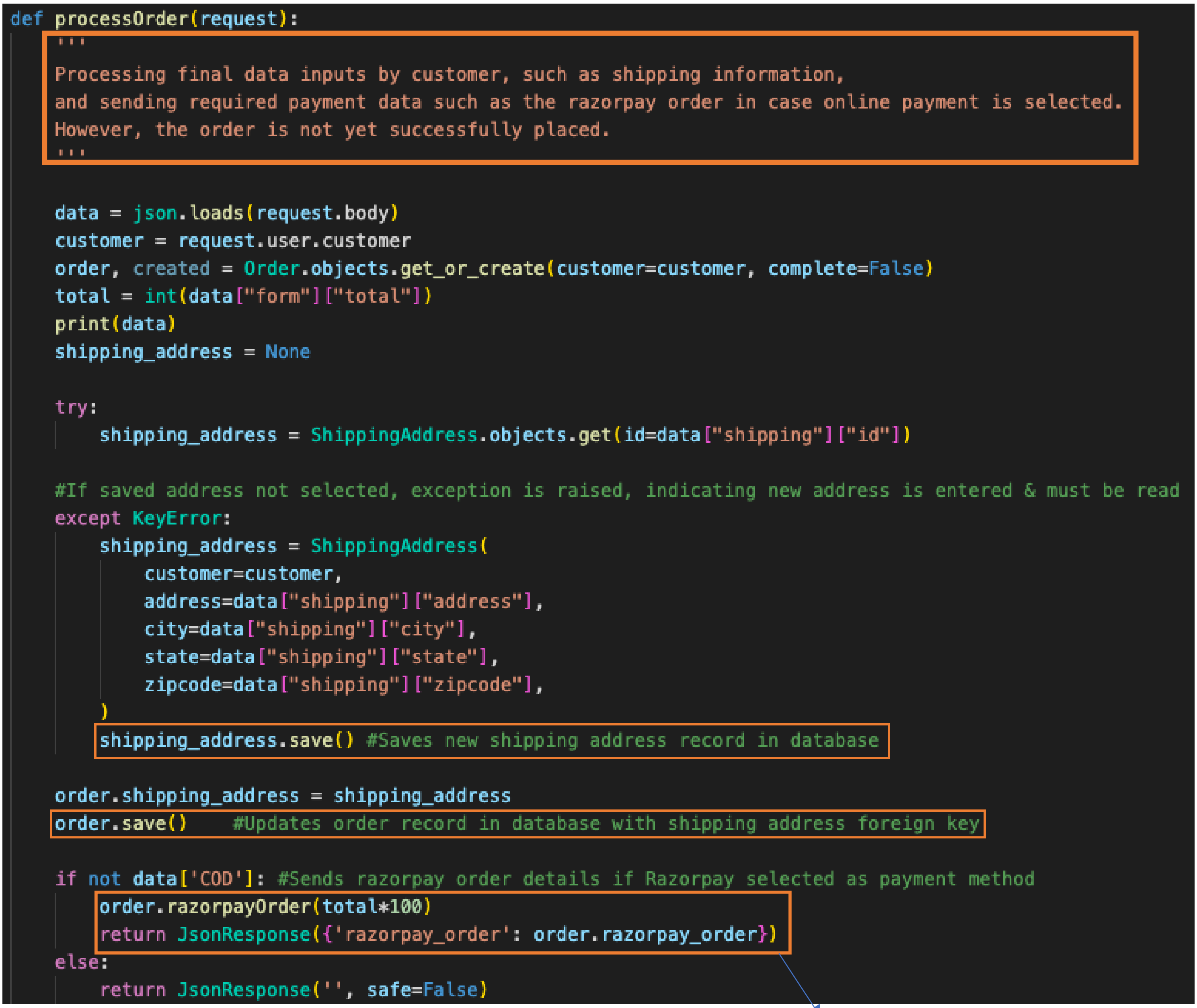
Data sent to backend for processing

COD flag determined by payment method chosen by customer (COD or online)



```
document.getElementById('razorpay').addEventListener('click', function(e){
    var COD = false;
    submitFormData(COD) //COD flag used to manage payment processing
})

document.getElementById('cod').addEventListener('click', function(e){
    var COD = true;
    submitFormData(COD)
})
```

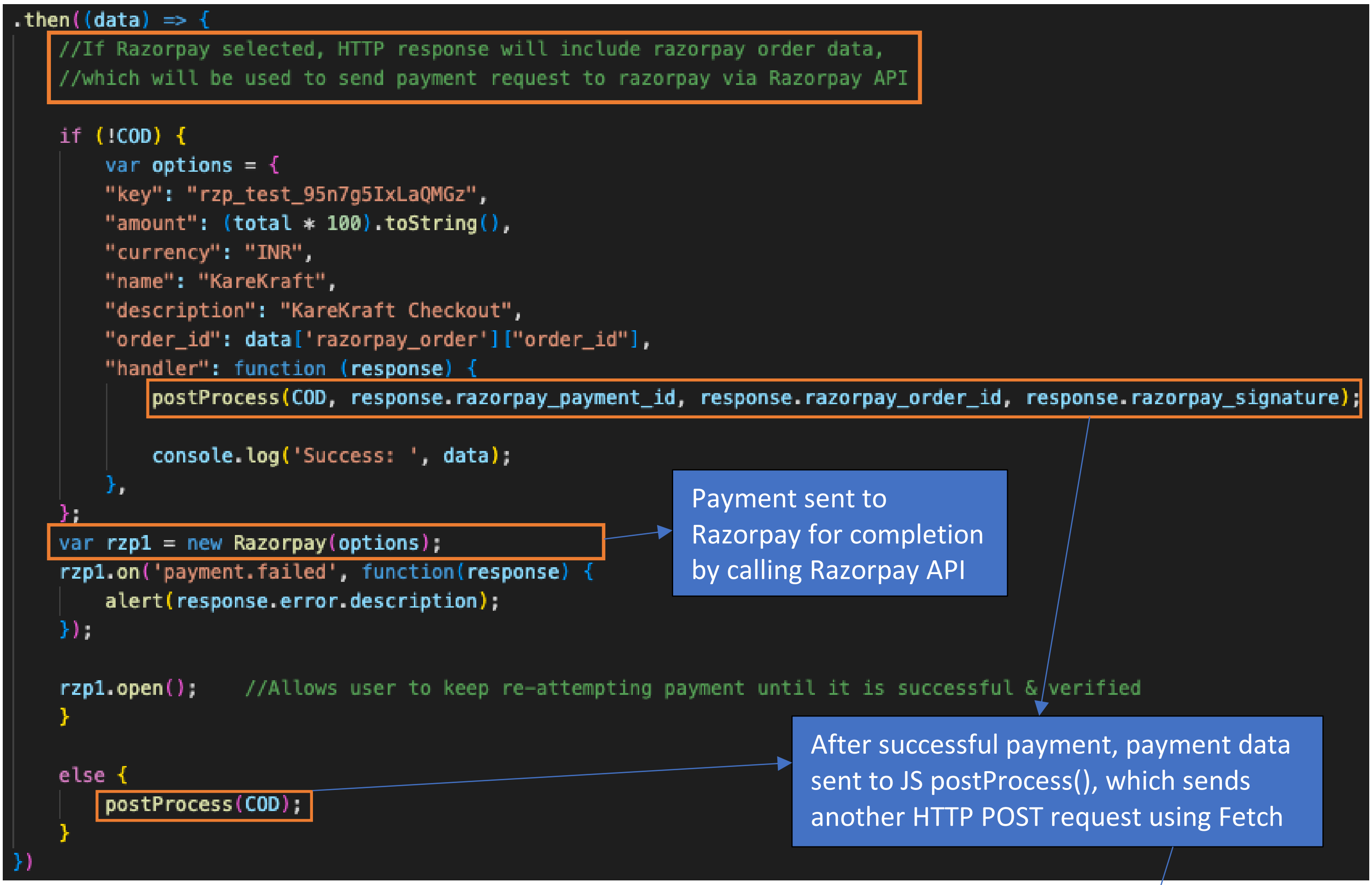
This data is read by processOrder(), which saves shipping details in the database and initiates the Razorpay payment (if online payment selected/flag COD = False)



```python
def processOrder(request):
    '''
    Processing final data inputs by customer, such as shipping information,
    and sending required payment data such as the razorpay order in case online payment is selected.
    However, the order is not yet successfully placed.
    '''

    data = json.loads(request.body)
    customer = request.user.customer
    order, created = Order.objects.get_or_create(customer=customer, complete=False)
    total = int(data["form"]["total"])
    print(data)
    shipping_address = None

    try:
        shipping_address = ShippingAddress.objects.get(id=data["shipping"]["id"])

    #If saved address not selected, exception is raised, indicating new address is entered & must be read
    except KeyError:
        shipping_address = ShippingAddress(
            customer=customer,
            address=data["shipping"]["address"],
            city=data["shipping"]["city"],
            state=data["shipping"]["state"],
            zipcode=data["shipping"]["zipcode"],
        )
        shipping_address.save() #Saves new shipping address record in database

    order.shipping_address = shipping_address
    order.save()    #Updates order record in database with shipping address foreign key

    if not data['COD']: #Sends razorpay order details if Razorpay selected as payment method
        order.razorpayOrder(total*100)
        return JsonResponse({'razorpay_order': order.razorpay_order})
    else:
        return JsonResponse('', safe=False)
```

Razorpay order created & sent to backend as HTTP response

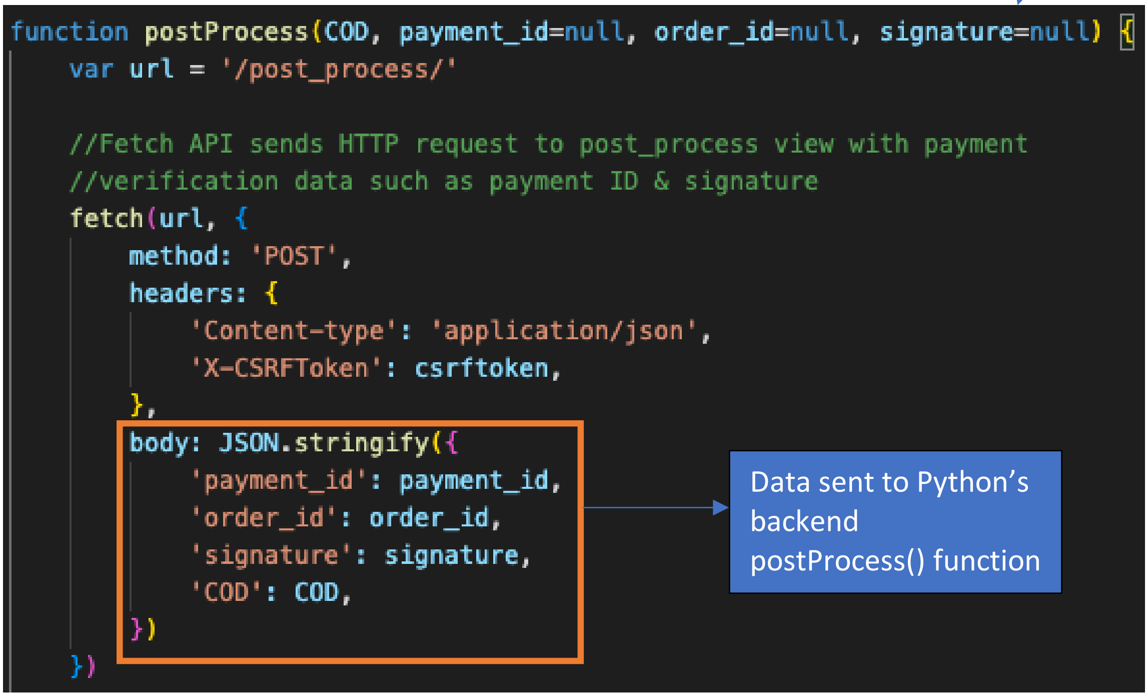
Razorpay Order data sent to backend is used in a payment request to Razorpay, made using the Razorpay API.



```
.then((data) => {
    //If Razorpay selected, HTTP response will include razorpay order data,
    //which will be used to send payment request to razorpay via Razorpay API

    if (!COD) {
        var options = {
        "key": "rzp_test_95n7g5IxLaQMGz",
        "amount": (total * 100).toString(),
        "currency": "INR",
        "name": "KareKraft",
        "description": "KareKraft Checkout",
        "order_id": data['razorpay_order']["order_id"],
        "handler": function (response) {
            postProcess(COD, response.razorpay_payment_id, response.razorpay_order_id, response.razorpay_signature);

            console.log('Success: ', data);
        },
    };
    var rzp1 = new Razorpay(options);
    rzp1.on('payment.failed', function(response) {
        alert(response.error.description);
    });

    rzp1.open();    //Allows user to keep re-attempting payment until it is successful & verified
    }

    else {
        postProcess(COD);
    }
})
```

Payment sent to Razorpay for completion by calling Razorpay API

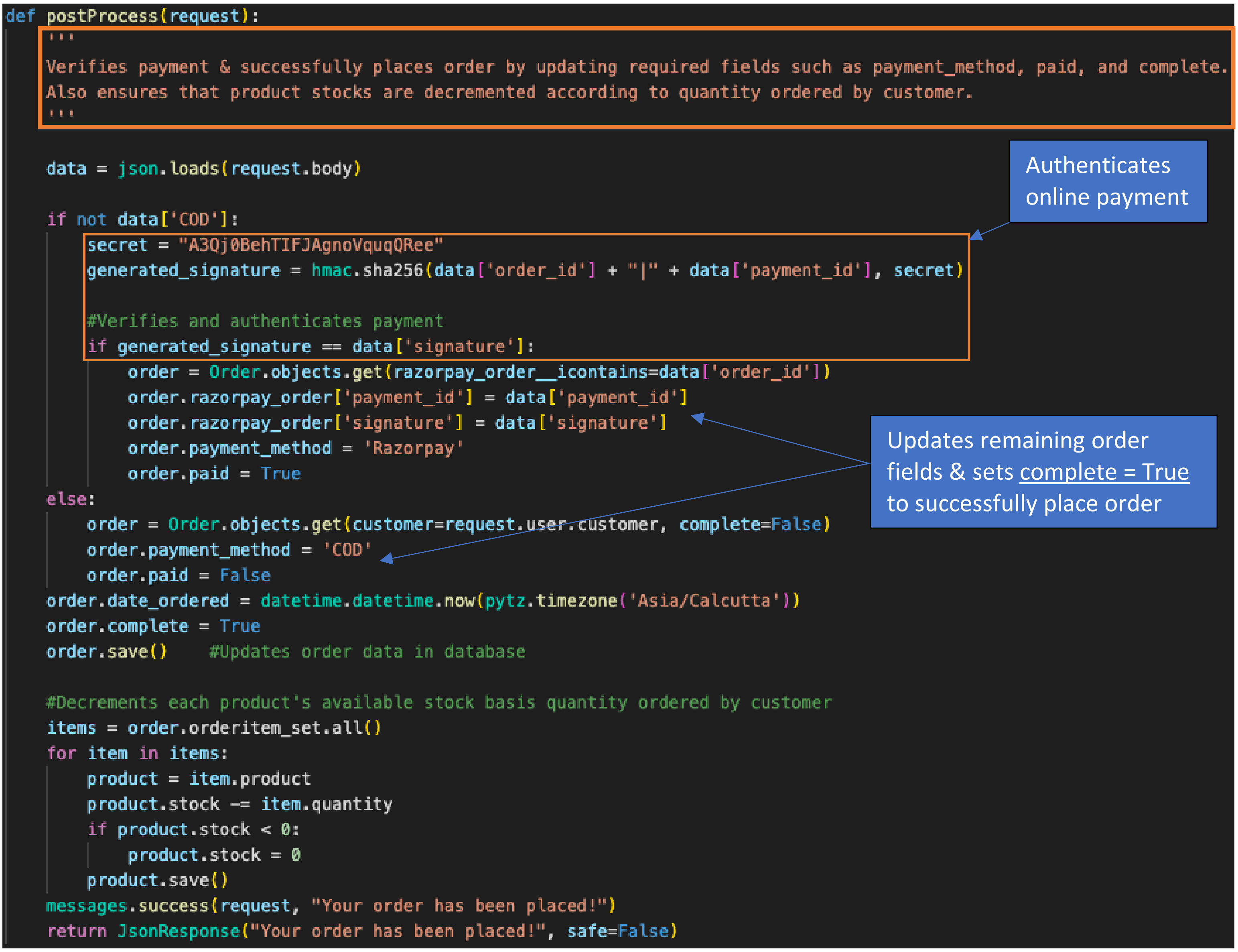
After successful payment, payment data sent to JS postProcess(), which sends another HTTP POST request using Fetch



```
function postProcess(COD, payment_id=null, order_id=null, signature=null) {
    var url = '/post_process/'

    //Fetch API sends HTTP request to post_process view with payment
    //verification data such as payment ID & signature
    fetch(url, {
        method: 'POST',
        headers: {
            'Content-type': 'application/json',
            'X-CSRFToken': csrftoken,
        },
        body: JSON.stringify({
            'payment_id': payment_id,
            'order_id': order_id,
            'signature': signature,
            'COD': COD,
        })
    })
```

Data sent to Python's backend postProcess() function

The postProcess() function then verifies online payment, successfully completes/places the order, and decrements available product stocks basis quantity ordered by the customer.



```
def postProcess(request):
    '''
    Verifies payment & successfully places order by updating required fields such as payment_method, paid, and complete.
    Also ensures that product stocks are decremented according to quantity ordered by customer.
    '''

    data = json.loads(request.body)

    if not data['COD']:
        secret = "A3Qj0BehTIFJAgnoVquqQRee"
        generated_signature = hmac.sha256(data['order_id'] + "|" + data['payment_id'], secret)

        #Verifies and authenticates payment
        if generated_signature == data['signature']:
            order = Order.objects.get(razorpay_order__icontains=data['order_id'])
            order.razorpay_order['payment_id'] = data['payment_id']
            order.razorpay_order['signature'] = data['signature']
            order.payment_method = 'Razorpay'
            order.paid = True
    else:
        order = Order.objects.get(customer=request.user.customer, complete=False)
        order.payment_method = 'COD'
        order.paid = False
    order.date_ordered = datetime.datetime.now(pytz.timezone('Asia/Calcutta'))
    order.complete = True
    order.save()    #Updates order data in database

    #Decrements each product's available stock basis quantity ordered by customer
    items = order.orderitem_set.all()
    for item in items:
        product = item.product
        product.stock -= item.quantity
        if product.stock < 0:
            product.stock = 0
        product.save()
    messages.success(request, "Your order has been placed!")
    return JsonResponse("Your order has been placed!", safe=False)
```

Authenticates online payment

Updates remaining order fields & sets complete = True to successfully place order

By breaking up this checkout procedure over multiple functions and HTTP requests, I have:

1. **Prevented errors** – If post-processing fails, then only the postProcess function is repeated, payment is not re-attempted. This prevents customers having to pay twice.

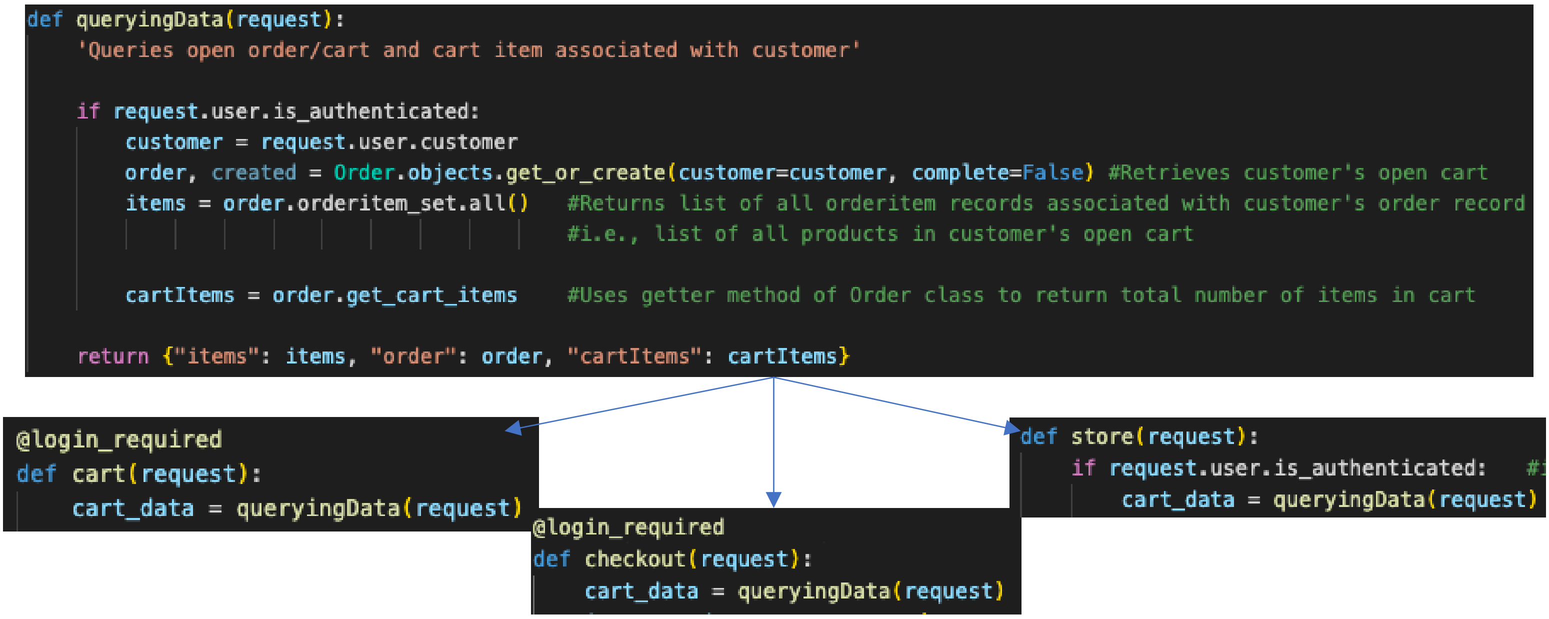
2. **Prevented data inconsistency** – If online payment fails and is repeated, the program does not try to save order/shipping address data again, preventing data inconsistencies.

3. **Facilitated Error Identification** – Breaking up the process makes it easy to identify where checkout failed and what data must be re-entered/re-processed etc.

4. **Facilitated Logical Functionality** – It is only logical to call an order 'complete' once I have received and verified that the payment has been completed.

Use of Functions

queryingData() – Same code used in homepage, search results, cart, and checkout – thus, placed in function and invoked to maintain a DRY methodology.



```
def queryingData(request):
    'Queries open order/cart and cart item associated with customer'

    if request.user.is_authenticated:
        customer = request.user.customer
        order, created = Order.objects.get_or_create(customer=customer, complete=False) #Retrieves customer's open cart
        items = order.orderitem_set.all()   #Returns list of all orderitem records associated with customer's order record
                                            #i.e., list of all products in customer's open cart

        cartItems = order.get_cart_items    #Uses getter method of Order class to return total number of items in cart

    return {"items": items, "order": order, "cartItems": cartItems}
```

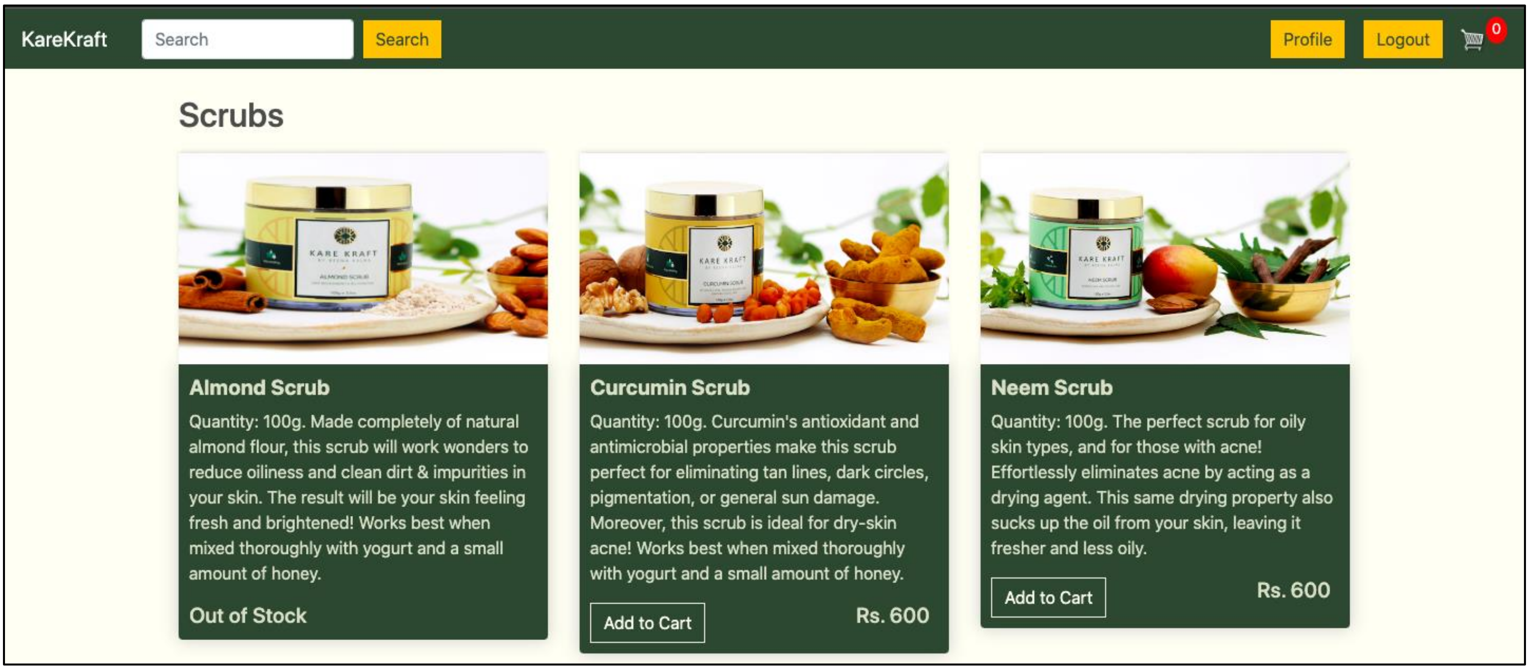
```
@login_required
def cart(request):
    cart_data = queryingData(request)
```

```
@login_required
def checkout(request):
    cart_data = queryingData(request)
```

```
def store(request):
    if request.user.is_authenticated:    #i
        cart_data = queryingData(request)
```

User Interface

1. **Login validations ensure only logged-in customers can buy; as per my product's scope.**



Scrubs

Almond Scrub

Quantity: 100g. Made completely of natural almond flour, this scrub will work wonders to reduce oiliness and clean dirt & impurities in your skin. The result will be your skin feeling fresh and brightened! Works best when mixed thoroughly with yogurt and a small amount of honey.

Out of Stock

Curcumin Scrub

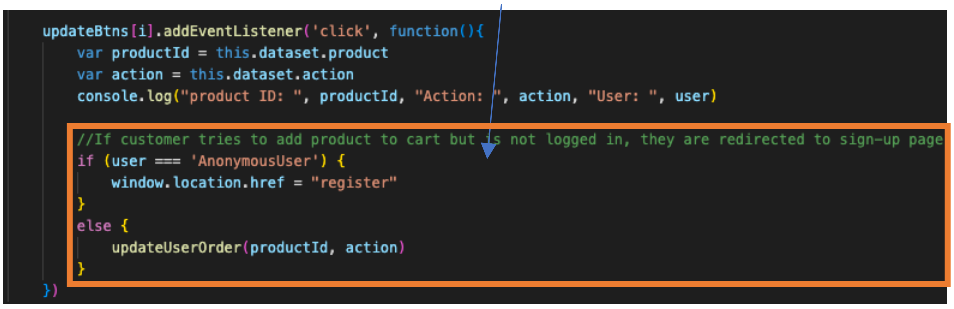
Quantity: 100g. Curcumin's antioxidant and antimicrobial properties make this scrub perfect for eliminating tan lines, dark circles, pigmentation, or general sun damage. Moreover, this scrub is ideal for dry-skin acne! Works best when mixed thoroughly with yogurt and a small amount of honey.

Add to Cart   Rs. 600

Neem Scrub

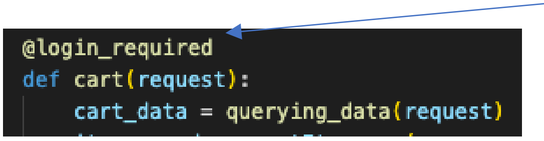
Quantity: 100g. The perfect scrub for oily skin types, and for those with acne! Effortlessly eliminates acne by acting as a drying agent. This same drying property also sucks up the oil from your skin, leaving it fresher and less oily.

Add to Cart   Rs. 600

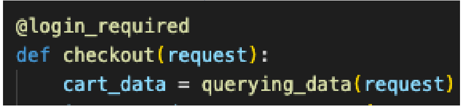


```
updateBtns[i].addEventListener('click', function(){
    var productId = this.dataset.product
    var action = this.dataset.action
    console.log("product ID: ", productId, "Action: ", action, "User: ", user)

    //If customer tries to add product to cart but is not logged in, they are redirected to sign-up page
    if (user === 'AnonymousUser') {
        window.location.href = "register"
    }
    else {
        updateUserOrder(productId, action)
    }
})
```

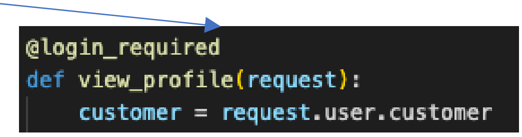
@login_required decorator automatically redirects non-logged-in users to registration page, even if they attempt to directly access cart/checkout/view profile URLs.



```
@login_required
def cart(request):
    cart_data = querying_data(request)
```



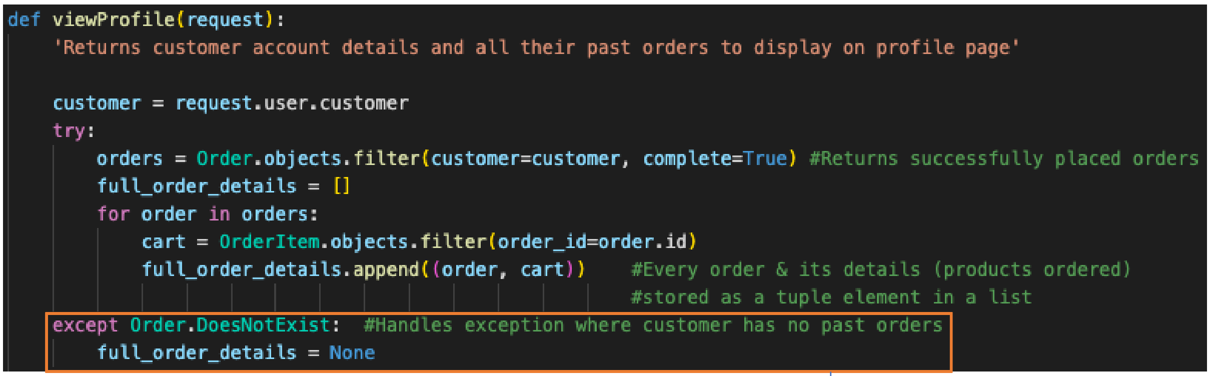
```
@login_required
def checkout(request):
    cart_data = querying_data(request)
```



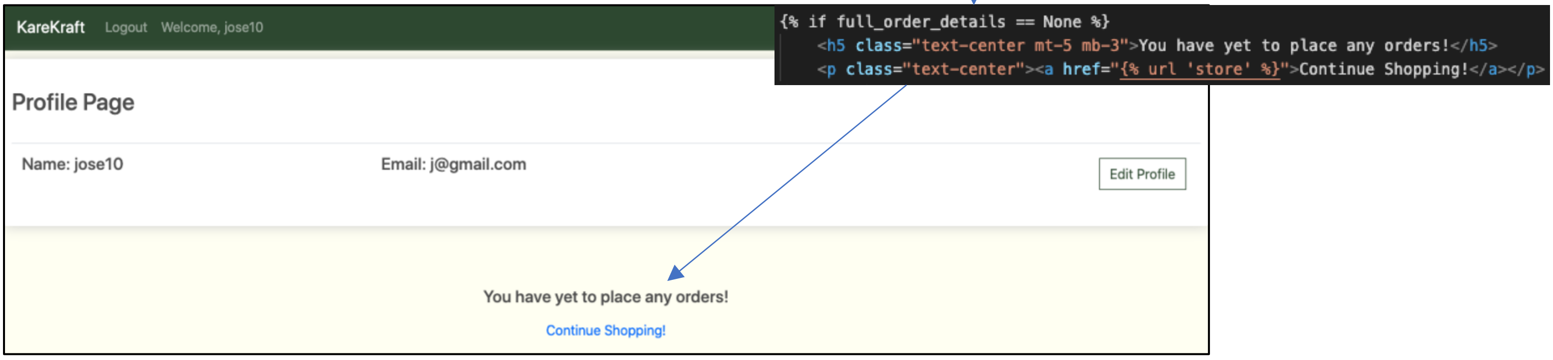
```
@login_required
def view_profile(request):
    customer = request.user.customer
```

2. **Interface modifications and try-except clauses used to handle errors/exceptions in user navigation.**

E.g., When viewing profile, try-except clause manages case where customer has no previous orders placed.



```
def viewProfile(request):
    'Returns customer account details and all their past orders to display on profile page'

    customer = request.user.customer
    try:
        orders = Order.objects.filter(customer=customer, complete=True) #Returns successfully placed orders
        full_order_details = []
        for order in orders:
            cart = OrderItem.objects.filter(order_id=order.id)
            full_order_details.append((order, cart))    #Every order & its details (products ordered)
                                                        #stored as a tuple element in a list
    except Order.DoesNotExist:  #Handles exception where customer has no past orders
        full_order_details = None
```



```
{% if full_order_details == None %}
    <h5 class="text-center mt-5 mb-3">You have yet to place any orders!</h5>
    <p class="text-center"><a href="{% url 'store' %}">Continue Shopping!</a></p>
```

Profile Page

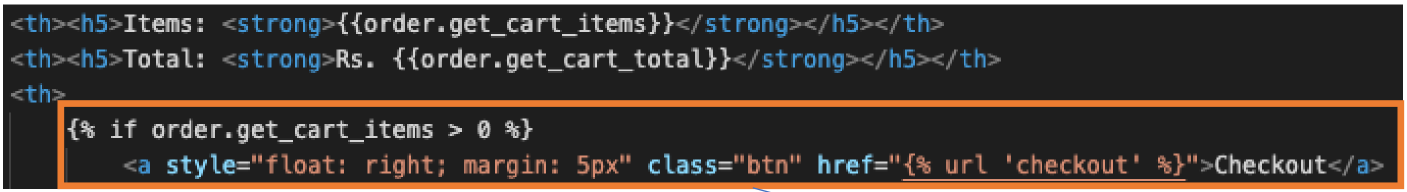
Name: jose10

Email: j@gmail.com

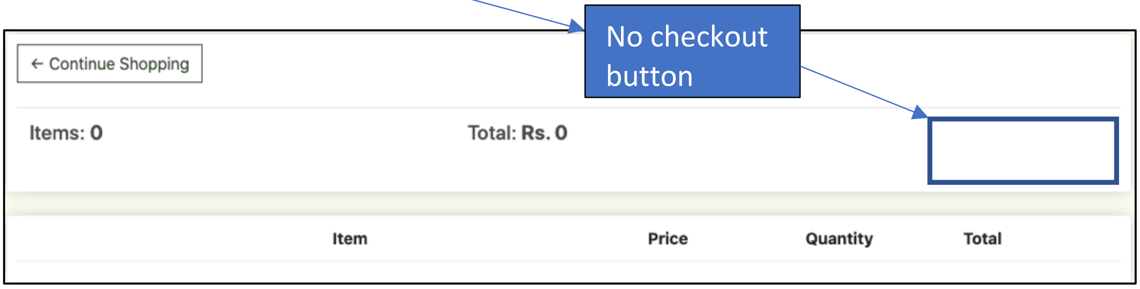
You have yet to place any orders!

Continue Shopping!

E.g., You cannot move from cart to checkout page if no items are in cart, preventing placement of empty orders.



```
<th><h5>Items: <strong>{{order.get_cart_items}}</strong></h5></th>
<th><h5>Total: <strong>Rs. {{order.get_cart_total}}</strong></h5></th>
<th>
    {% if order.get_cart_items > 0 %}
        <a style="float: right; margin: 5px" class="btn" href="{% url 'checkout' %}">Checkout</a>
```



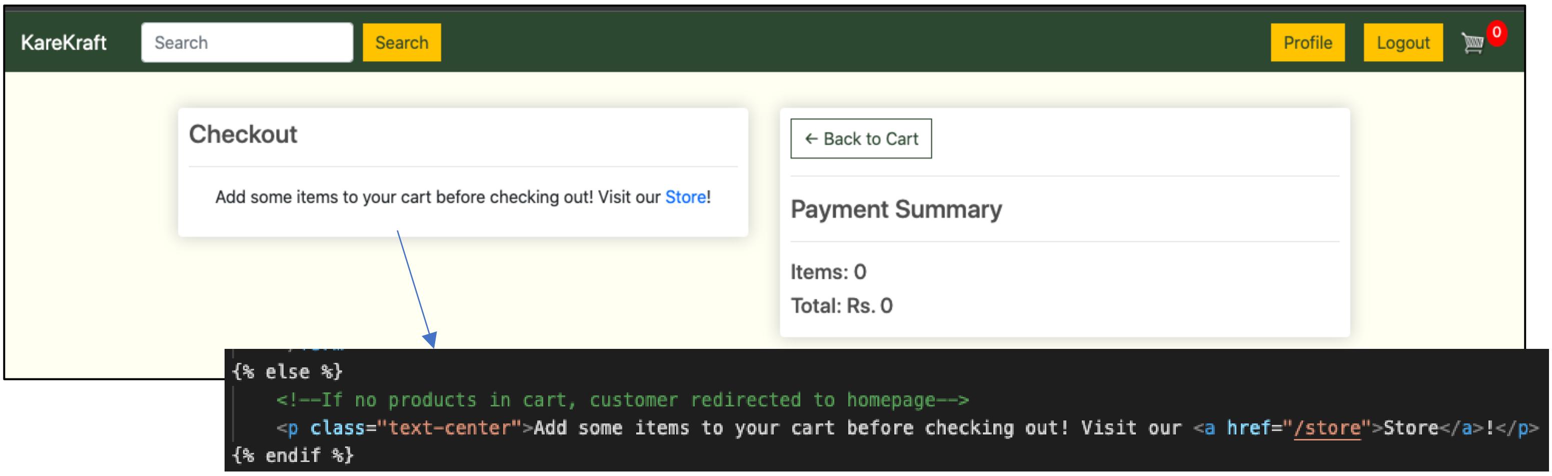
No checkout button

Items: 0

Total: Rs. 0

Item	Price	Quantity	Total

Moreover, if customer with empty cart then attempts to directly access checkout page via URL, they still cannot place the order:



Checkout

Add some items to your cart before checking out! Visit our Store!

← Back to Cart

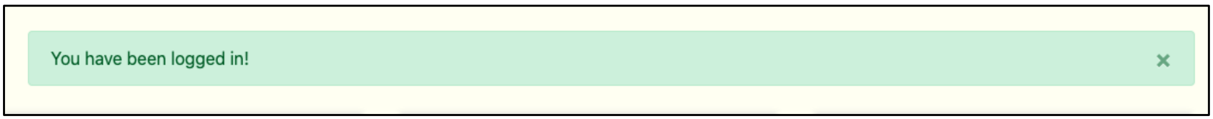
Payment Summary

Items: 0

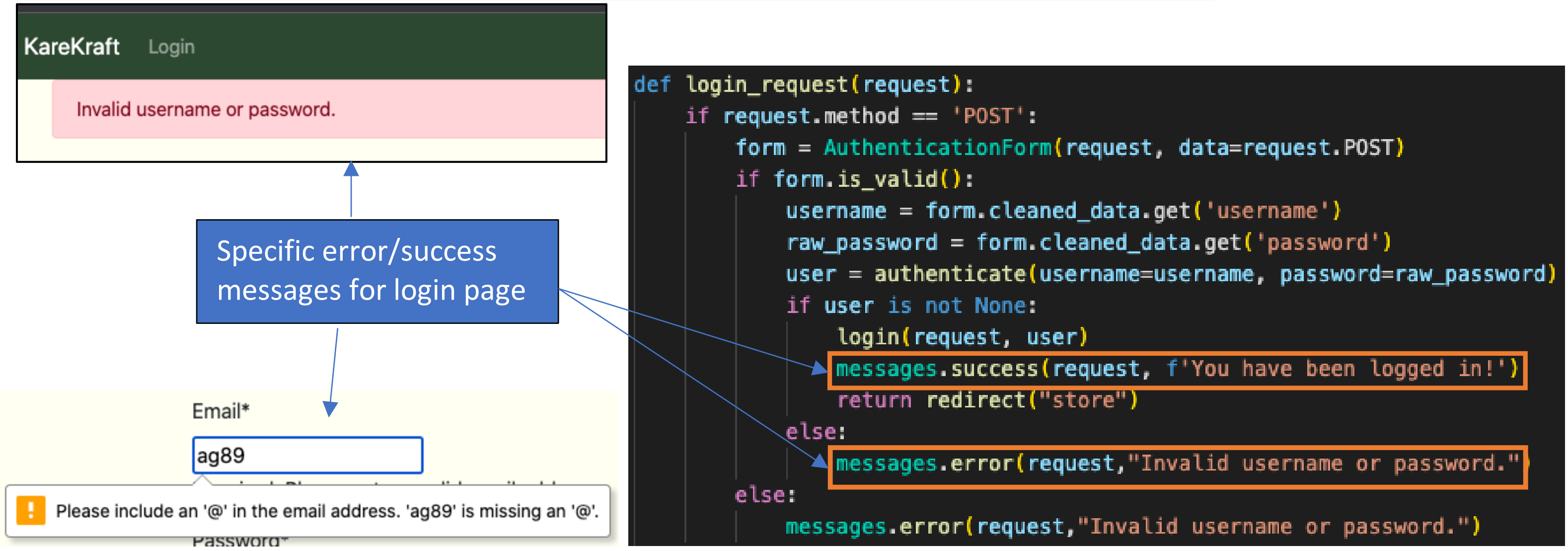
Total: Rs. 0

```
{% else %}
    <!--If no products in cart, customer redirected to homepage-->
    <p class="text-center">Add some items to your cart before checking out! Visit our <a href="/store">Store</a>!</p>
{% endif %}
```

3. Use of appropriate alerts to give users success/error/warning messages.



You have been logged in! ×



Invalid username or password.

Specific error/success messages for login page

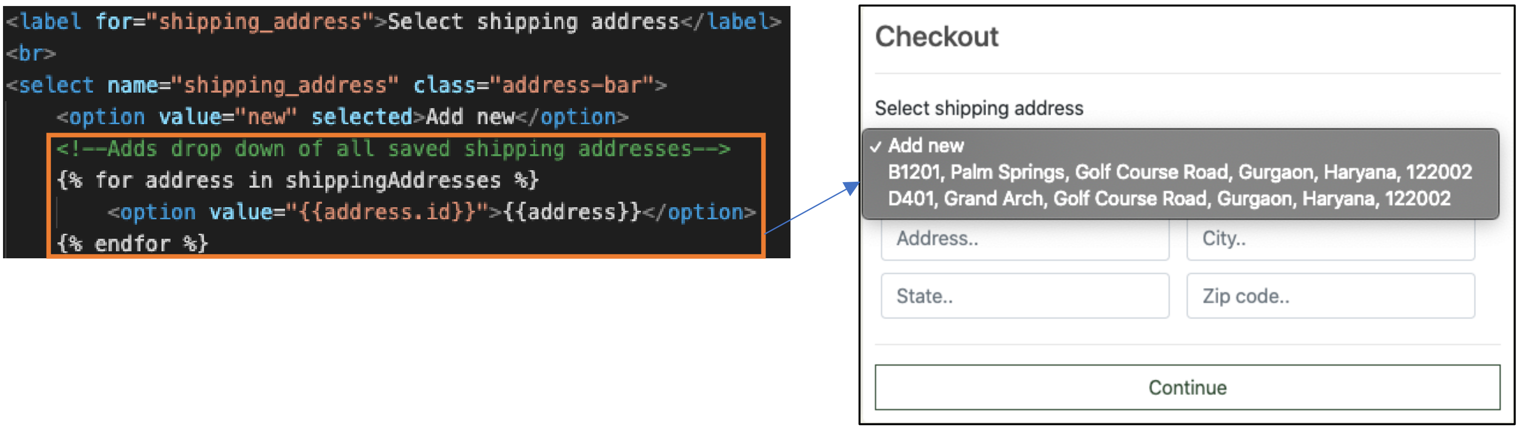
Email*

ag89

Please include an '@' in the email address. 'ag89' is missing an '@'.

```
def login_request(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            username = form.cleaned_data.get('username')
            raw_password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=raw_password)
            if user is not None:
                login(request, user)
                messages.success(request, f'You have been logged in!')
                return redirect("store")
            else:
                messages.error(request,"Invalid username or password.")
        else:
            messages.error(request,"Invalid username or password.")
```

4. Every new shipping address entered by the customer is saved, and can be reused in future orders, thus minimizing customer text entry.

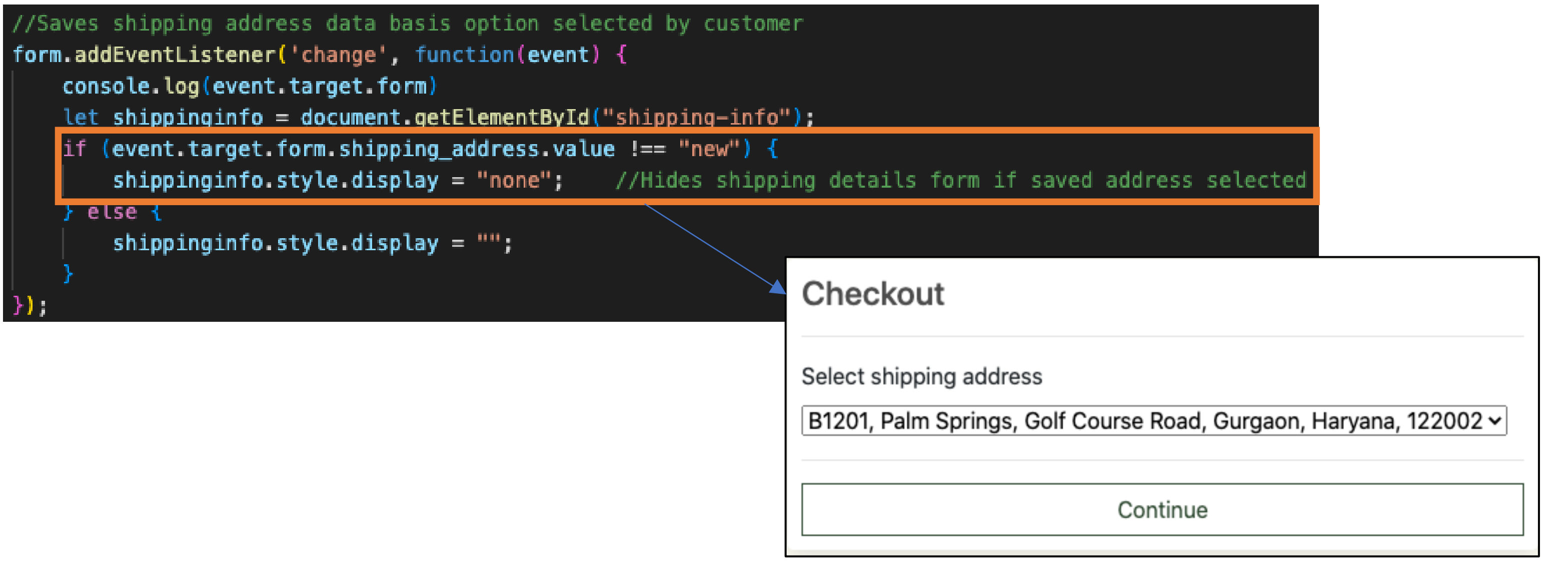


```
<label for="shipping_address">Select shipping address</label>
<br>
<select name="shipping_address" class="address-bar">
    <option value="new" selected>Add new</option>
    <!--Adds drop down of all saved shipping addresses-->
    {% for address in shippingAddresses %}
        <option value="{{address.id}}">{{address}}</option>
    {% endfor %}
```

Checkout

Select shipping address

✓ Add new
B1201, Palm Springs, Golf Course Road, Gurgaon, Haryana, 122002
D401, Grand Arch, Golf Course Road, Gurgaon, Haryana, 122002



```
//Saves shipping address data basis option selected by customer
form.addEventListener('change', function(event) {
    console.log(event.target.form)
    let shippinginfo = document.getElementById("shipping-info");
    if (event.target.form.shipping_address.value !== "new") {
        shippinginfo.style.display = "none";    //Hides shipping details form if saved address selected
    } else {
        shippinginfo.style.display = "";
    }
});
```

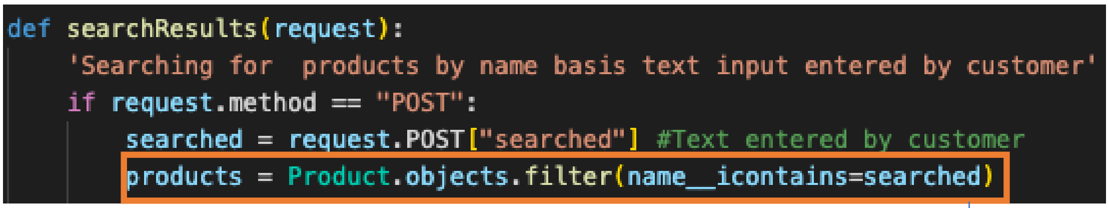
Checkout

Select shipping address

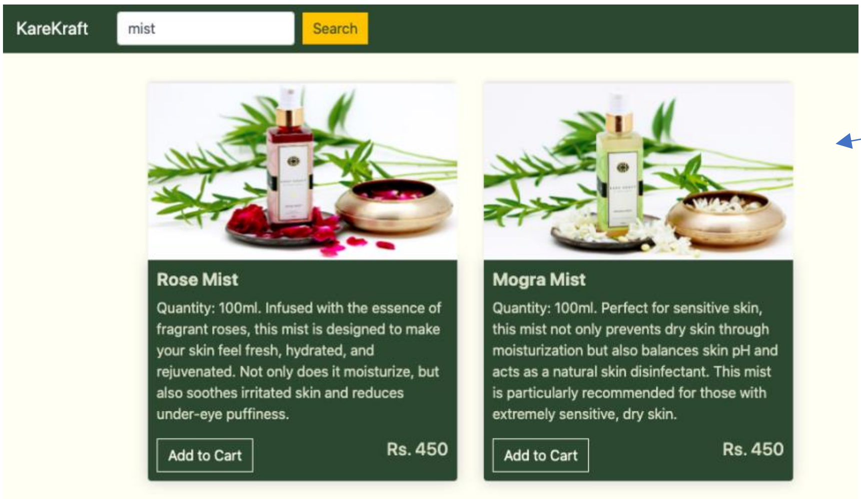
B1201, Palm Springs, Golf Course Road, Gurgaon, Haryana, 122002

Searching and Filtering

Customers can search for products basis their names, using **‘name__icontains’**. This does a case-insensitive search using wildcards, similar to **LIKE** or **%%** in SQL.



```
def searchResults(request):
    'Searching for  products by name basis text input entered by customer'
    if request.method == "POST":
        searched = request.POST["searched"] #Text entered by customer
        products = Product.objects.filter(name__icontains=searched)
```



Rose Mist

Quantity: 100ml. Infused with the essence of fragrant roses, this mist is designed to make your skin feel fresh, hydrated, and rejuvenated. Not only does it moisturize, but also soothes irritated skin and reduces under-eye puffiness.

Rs. 450

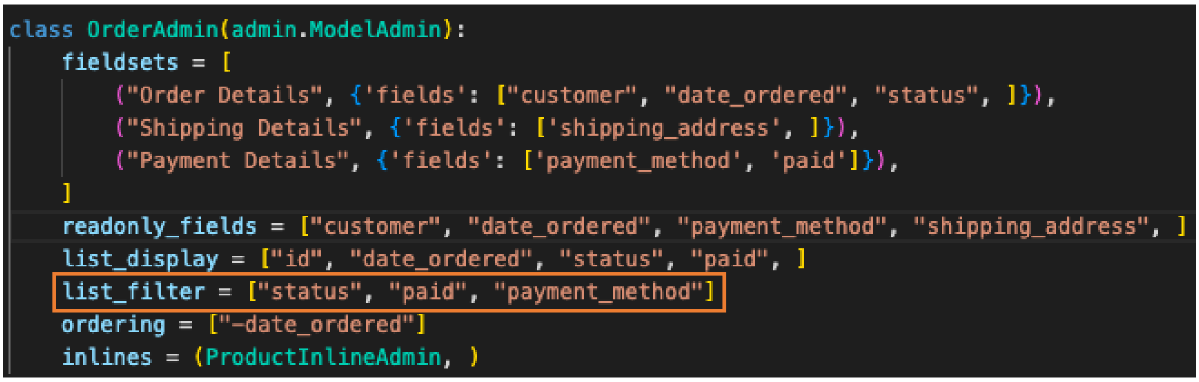
Mogra Mist

Quantity: 100ml. Perfect for sensitive skin, this mist not only prevents dry skin through moisturization but also balances skin pH and acts as a natural skin disinfectant. This mist is particularly recommended for those with extremely sensitive, dry skin.

Rs. 450

Relevant results output in same style as homepage

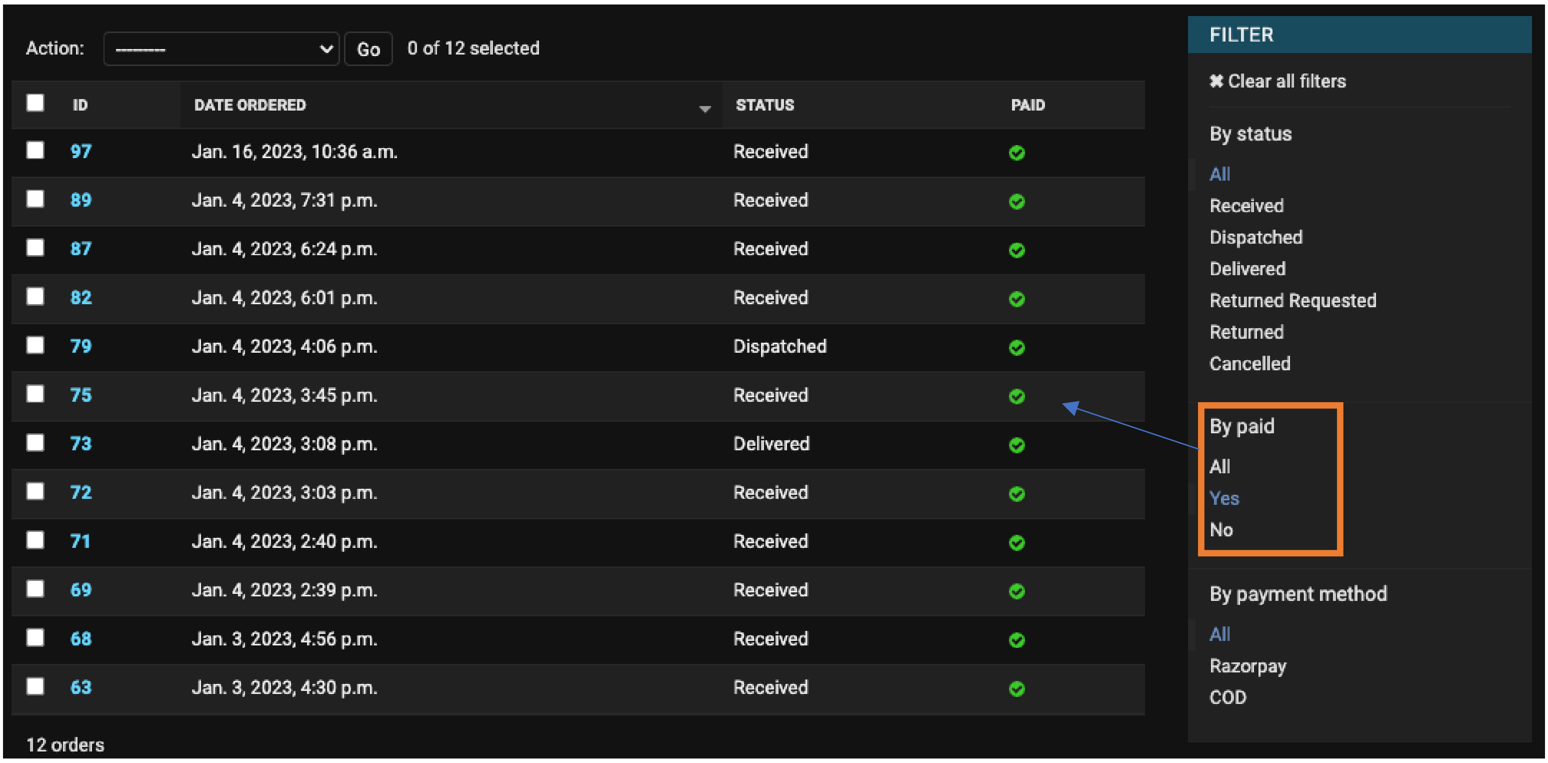
Administrators can also filter through their orders basis parameters defined in **‘list_filter’**.



```
class OrderAdmin(admin.ModelAdmin):
    fieldsets = [
        ("Order Details", {'fields': ["customer", "date_ordered", "status", ]}),
        ("Shipping Details", {'fields': ['shipping_address', ]}),
        ("Payment Details", {'fields': ['payment_method', 'paid']}),
    ]
    readonly_fields = ["customer", "date_ordered", "payment_method", "shipping_address", ]
    list_display = ["id", "date_ordered", "status", "paid", ]
    list_filter = ["status", "paid", "payment_method"]
    ordering = ["-date_ordered"]
    inlines = (ProductInlineAdmin, )
```

When a specific filter is chosen, e.g., when filter = paid and 'True' selected, orders queried using:

SELECT order WHERE paid == True



Action: -------- Go 0 of 12 selected

☐	ID	DATE ORDERED	STATUS	PAID
☐	97	Jan. 16, 2023, 10:36 a.m.	Received	✓
☐	89	Jan. 4, 2023, 7:31 p.m.	Received	✓
☐	87	Jan. 4, 2023, 6:24 p.m.	Received	✓
☐	82	Jan. 4, 2023, 6:01 p.m.	Received	✓
☐	79	Jan. 4, 2023, 4:06 p.m.	Dispatched	✓
☐	75	Jan. 4, 2023, 3:45 p.m.	Received	✓
☐	73	Jan. 4, 2023, 3:08 p.m.	Delivered	✓
☐	72	Jan. 4, 2023, 3:03 p.m.	Received	✓
☐	71	Jan. 4, 2023, 2:40 p.m.	Received	✓
☐	69	Jan. 4, 2023, 2:39 p.m.	Received	✓
☐	68	Jan. 3, 2023, 4:56 p.m.	Received	✓
☐	63	Jan. 3, 2023, 4:30 p.m.	Received	✓

12 orders

FILTER

✖ Clear all filters

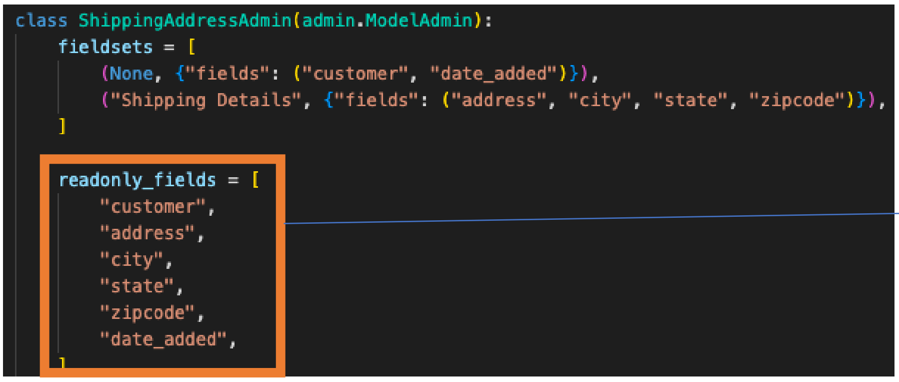
By status
- All
- Received
- Dispatched
- Delivered
- Returned Requested
- Returned
- Cancelled

By paid
- All
- Yes
- No

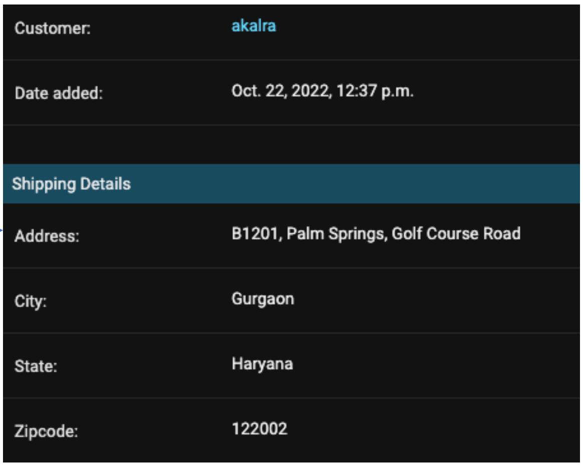
By payment method
- All
- Razorpay
- COD

Access Rights and Data Security

Access rights managed for administrators to ensure they cannot edit certain fields. E.g., shipping details of customer are read-only (defined under **readonly_fields**).



```
class ShippingAddressAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {"fields": ("customer", "date_added")}),
        ("Shipping Details", {"fields": ("address", "city", "state", "zipcode")}),
    ]

    readonly_fields = [
        "customer",
        "address",
        "city",
        "state",
        "zipcode",
        "date_added",
    ]
```



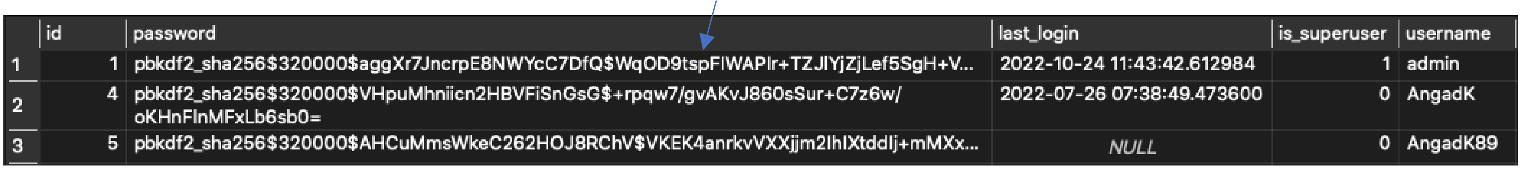
Customer:	akalra
Date added:	Oct. 22, 2022, 12:37 p.m.

Shipping Details

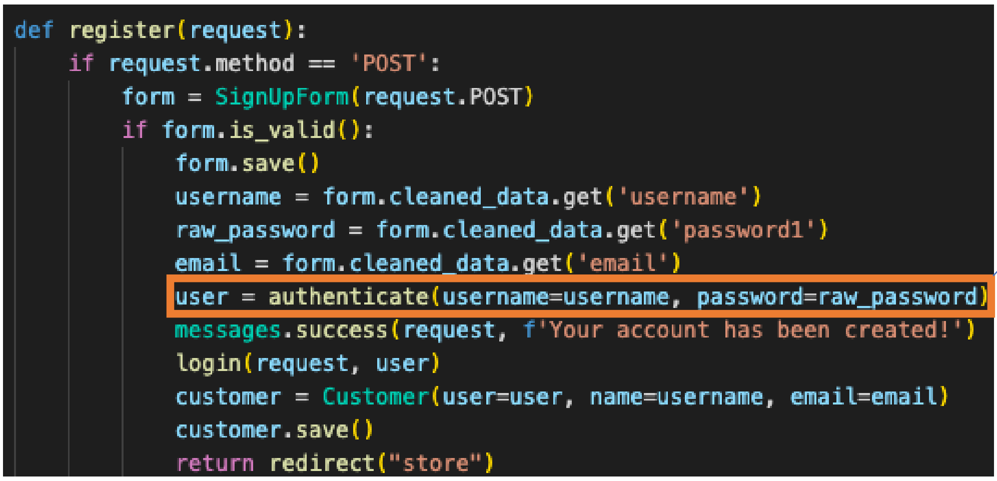
Address:	B1201, Palm Springs, Golf Course Road
City:	Gurgaon
State:	Haryana
Zipcode:	122002

Customer Table

Raw passwords not stored in database to maintain customer security. Rather, they are hashed before storage (raw passwords cannot be derived from hash).



	id	password	last_login	is_superuser	username
1	1	pbkdf2_sha256$320000$aggXr7JncrpE8NWYcC7DfQ$WqOD9tspFIWAPIr+TZJlYjZjLef5SgH+V...	2022-10-24 11:43:42.612984	1	admin
2	4	pbkdf2_sha256$320000$VHpuMhniicn2HBVFiSnGsG$+rpqw7/gvAKvJ860sSur+C7z6w/oKHnFlnMFxLb6sb0=	2022-07-26 07:38:49.473600	0	AngadK
3	5	pbkdf2_sha256$320000$AHCuMmsWkeC262HOJ8RChV$VKEK4anrkvVXXjjm2lhlXtddlj+mMXx...	NULL	0	AngadK89

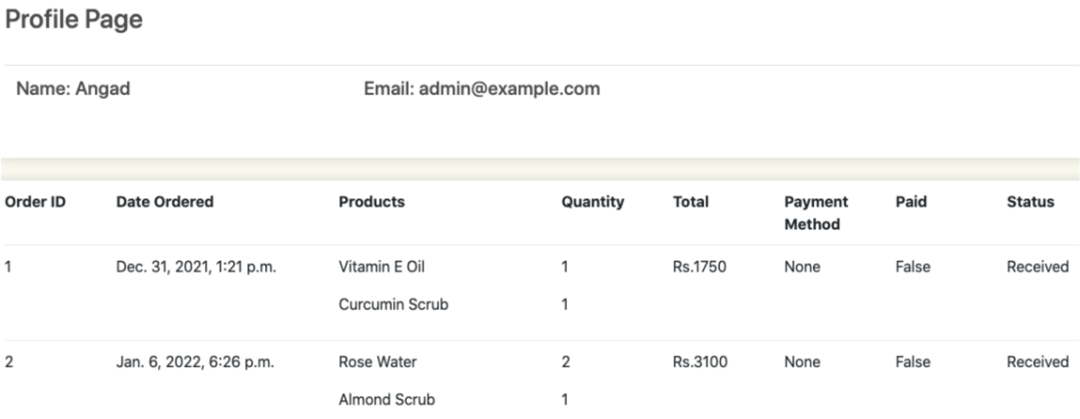


```
def register(request):
    if request.method == 'POST':
        form = SignUpForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username')
            raw_password = form.cleaned_data.get('password1')
            email = form.cleaned_data.get('email')
            user = authenticate(username=username, password=raw_password)
            messages.success(request, f'Your account has been created!')
            login(request, user)
            customer = Customer(user=user, name=username, email=email)
            customer.save()
            return redirect("store")
```

Password hashed by Django's "authenticate" function when signing up

Nested Loops

Use of nested loops on the view profile page to output the list of all products in each order placed by a user.

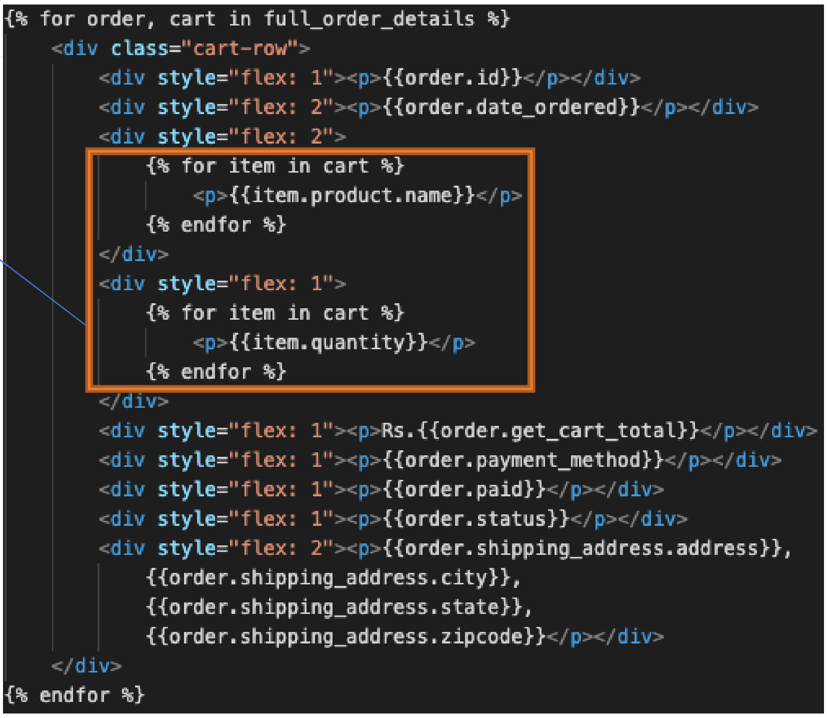


Profile Page

Name: Angad Email: admin@example.com

Order ID	Date Ordered	Products	Quantity	Total	Payment Method	Paid	Status
1	Dec. 31, 2021, 1:21 p.m.	Vitamin E Oil	1	Rs.1750	None	False	Received
		Curcumin Scrub	1				
2	Jan. 6, 2022, 6:26 p.m.	Rose Water	2	Rs.3100	None	False	Received
		Almond Scrub	1				
		Neem Scrub	1				

Nested for loop used to output all products & their quantities in the order



```
{% for order, cart in full_order_details %}
    <div class="cart-row">
        <div style="flex: 1"><p>{{order.id}}</p></div>
        <div style="flex: 2"><p>{{order.date_ordered}}</p></div>
        <div style="flex: 2">
            {% for item in cart %}
                <p>{{item.product.name}}</p>
            {% endfor %}
        </div>
        <div style="flex: 1">
            {% for item in cart %}
                <p>{{item.quantity}}</p>
            {% endfor %}
        </div>
        <div style="flex: 1"><p>Rs.{{order.get_cart_total}}</p></div>
        <div style="flex: 1"><p>{{order.payment_method}}</p></div>
        <div style="flex: 1"><p>{{order.paid}}</p></div>
        <div style="flex: 1"><p>{{order.status}}</p></div>
        <div style="flex: 2"><p>{{order.shipping_address.address}},
            {{order.shipping_address.city}},
            {{order.shipping_address.state}},
            {{order.shipping_address.zipcode}}</p></div>
    </div>
{% endfor %}
```

Word Count: 1126 words

Resources Used (MLA 8)

1. *Django Documentation*. Django, https://docs.djangoproject.com/en/4.1/. Accessed 8 July 2022.

2. “HTML Tutorial.” *W3Schools*, https://www.w3schools.com/html/default.asp. Accessed 23 July 2022.

3. “Integrate with Standard Web Checkout.” *Razorpay*, Razorpay Docs, https://razorpay.com/docs/payments/payment-gateway/web-integration/standard. Accessed 26 June 2022.

4. “Using the Fetch API.” *Web APIs | MDN*, MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. Accessed 12 June 2022.