Criteria B – Planning

Scope of Product

My product is an e-commerce website that will be used to sell the products of my client's brand, KareKraft.

The website's scope will be limited to only allow users who create an account and log in to shop. These logged in customers will enjoy the following functionalities:

- Being able to add multiple products with ≥ 1 quantity of each product into their cart
- Entering and saving their shipping address onto the website
- Paying for the order and successfully placing it

It must be noted here that orders placed through the website can only be for delivery addresses in India. Furthermore, payment will also only be allowed from domestic means (I.e., domestic credit/debit cards) only. Therefore, the product is targeted at customers of KareKraft located in India[1].

Adding on, my client has defined the scope of payment methods to include[2]:

- Cash on Delivery
- Credit/Debit cards
- Domestic UPIs like Paytm and Google Pay

As for website administrators, such as my client, their scope of functionality will include:

- Adding/updating the products that are available for sale on the website
- Viewing orders placed by customers
- Updating the order/payment statuses of an order

Design Specifications

Specifications	Details
Hardware Requirements	System Development: • **Device:** MacBook Air (Early 2015 model) • **Processor:** 1.6 GHz Dual-Core Intel Core i5 • **Memory:** 4GB RAM, 1600MHz DDR3 User Requirements: • Website's frontend display is optimized and designed for desktop/laptop use, though it can be scaled to fit a phone as well
Software Requirements	System Development: • **Operating System:** MacOS Catalina, Version 10.15.5

[1] Refer to Appendix – "Criteria B: Target Audience"
[2] Refer to Appendix – "Criteria B: Payment Methods"

	• **IDE and Version Used:** Visual Studio Code 1.68.1 (Universal) User Requirements: • The website should be compatible across different web platforms/search engines (E.g., Google, Microsoft Edge, Bing etc.)
Data Processing	For the project, the database will locally be implemented using SQLite3 as the database management system. Within the product, data will be processed in the following ways: 1. **Creation of database objects** o Creating new customer records o Creating new open orders 2. **Reading data from database into backend** o Reading available products into store o Reading data of orders placed by customers to display for administrator o Reading customer data when logging them in and displaying customer profile 3. **Updating data in database** o Updating customer details o Updating cart items (products and quantities) o Updating order details in backend (shipping details, payment details, status etc.) o Updating stock of available products 4. **Deleting data in database** o Deleting products from customer's cart o Deleting products from the website (administrator action)
Data Security	1. **Password Security:** Hashing passwords before storing them to maintain security of the customer's passwords 2. **Payment Security:** Ensuring that online payments are secure to protect customer's financial data
Data Integrity	1. **Managing extreme data:** Positive integer constraint on fields such as prices and available stock to prevent extreme data entry 2. **Managing abnormal data**: Types defined for each field (E.g., integer, string etc.), which return error messages in case of abnormal data entry 3. **Null entries:** 'Not Null' constraints placed on fields where data is necessary to prevent integrity issues due to missing/incomplete data (E.g., customer account details)
System Processing	The system will have the following functionalities: 1. Displaying the available products to users

	2. Displaying customer profile and previous order information on frontend 3. Displaying the customer's cart and allowing them to update cart items 4. Reading user inputs when creating customer accounts, updating profiles, entering shipping addresses for orders 5. Tracking payments of orders 6. Allowing the administrator (my client) to view and update specific order details (E.g., order status) for customers

User Interactions

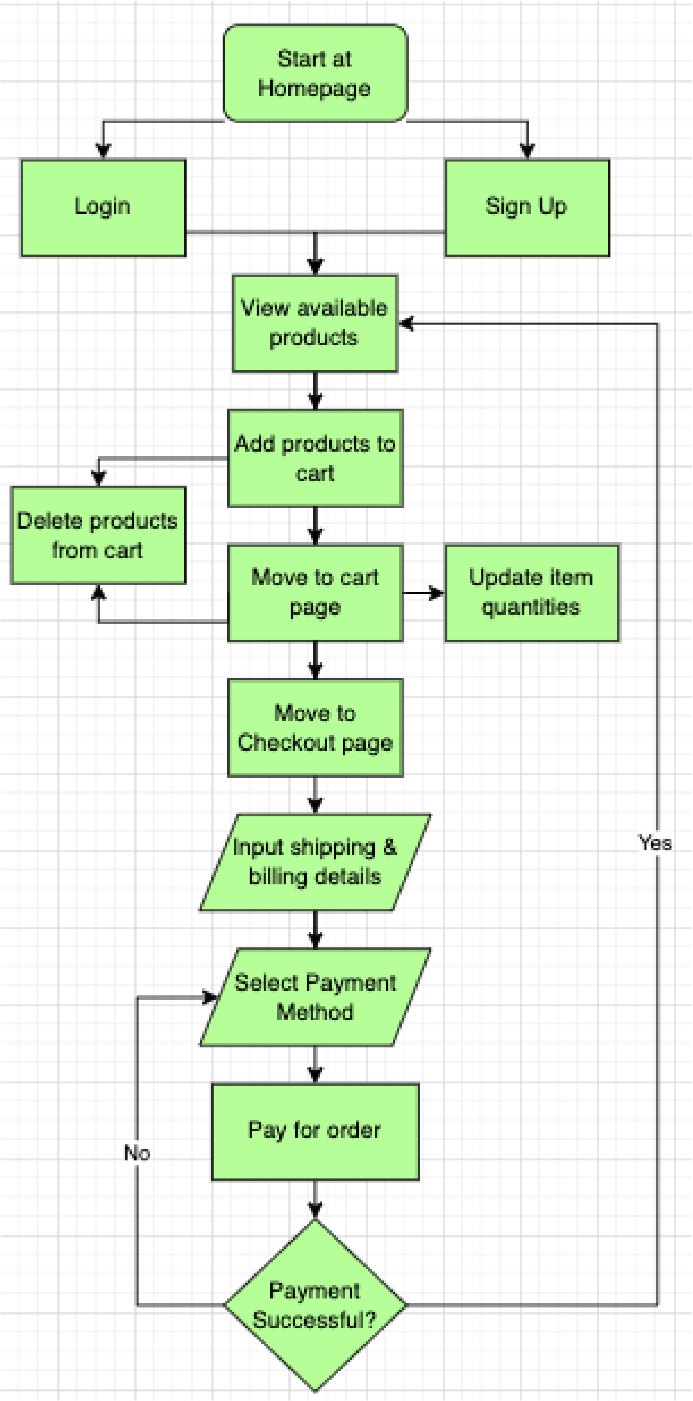
Users	Access Rights and Restrictions	Functionality Rights
Customer	**Accessible:** Customer-end website **Restricted:** Admin site, website backend, backend database	1. Viewing available products on website homepage 2. Registering a new account or logging into an existing account 3. Viewing and modifying existing account details 4. Adding/removing products to cart and viewing cart 5. Inputting shipping information for order 6. Paying for order using online means (credit/debit cards, UPI etc.) or via cash-on-delivery
Website Administrator	**Accessible:** Customer-end website, Admin site **Restricted:** Website backend, backend database	On the customer-end website, the administrator would have the same rights as a customer. On the admin site: 1. Login using administrator credentials 2. Add/remove products that are in stock, or update available quantities of already listed products 3. View orders placed by customers 4. View details of customer who has placed the order (i.e., name, contact information, and shipping address)

		5. Add/Remove administrator accounts and manage their access rights
Website Developer	**Accessible:** Customer-end website, admin site, website backend, backend database	On the customer-end website, the administrator would have the same rights as a customer. On the admin site, the developer would only view-only rights, and would not be allowed to edit any data on the site. As for the website backend and database, the developer is in-charge of: 1. Adding new features into the customer-end and admin sites, while ensuring appropriate system functionality and data processing 2. Appropriately designing the frontend view of the customer and admin sites basis approved design ideas 3. Managing the physical and logical integrity and security of data in the database 4. Ensuring efficient physical and logical storage of data, e.g., using normalization

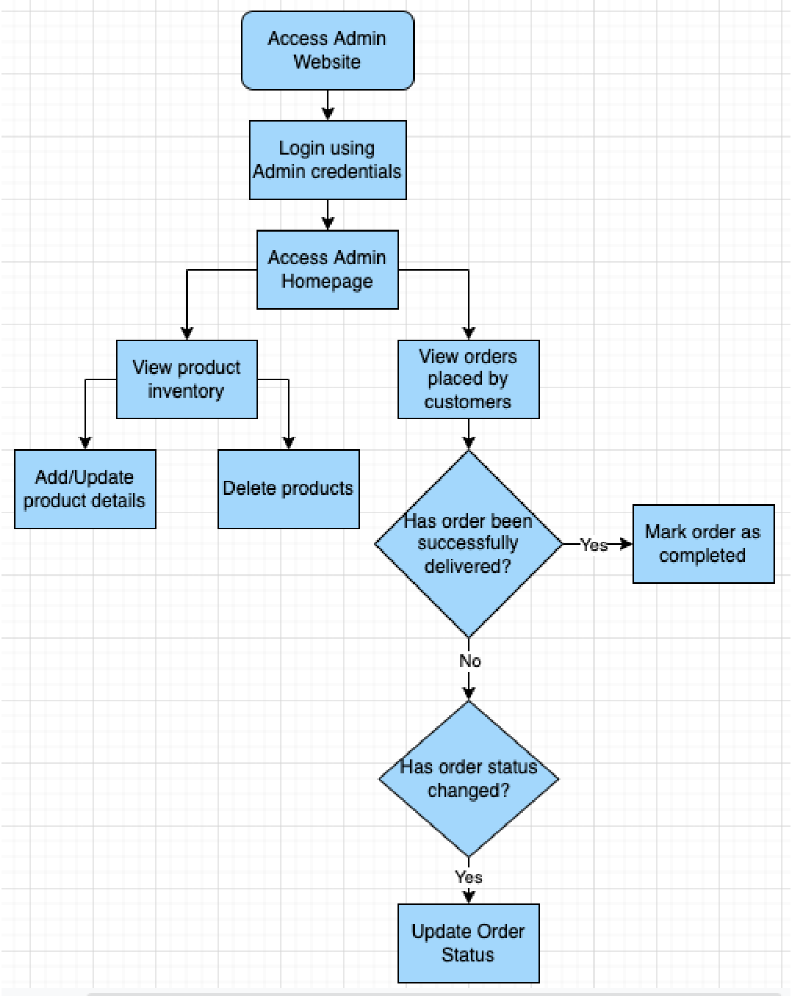
Typical Navigation Process

Before moving forward with the planning process, I would like to first define and understand the typical navigation process that both a client and administrator of the website will use. These will be as follows:

Typical Customer Navigation Process



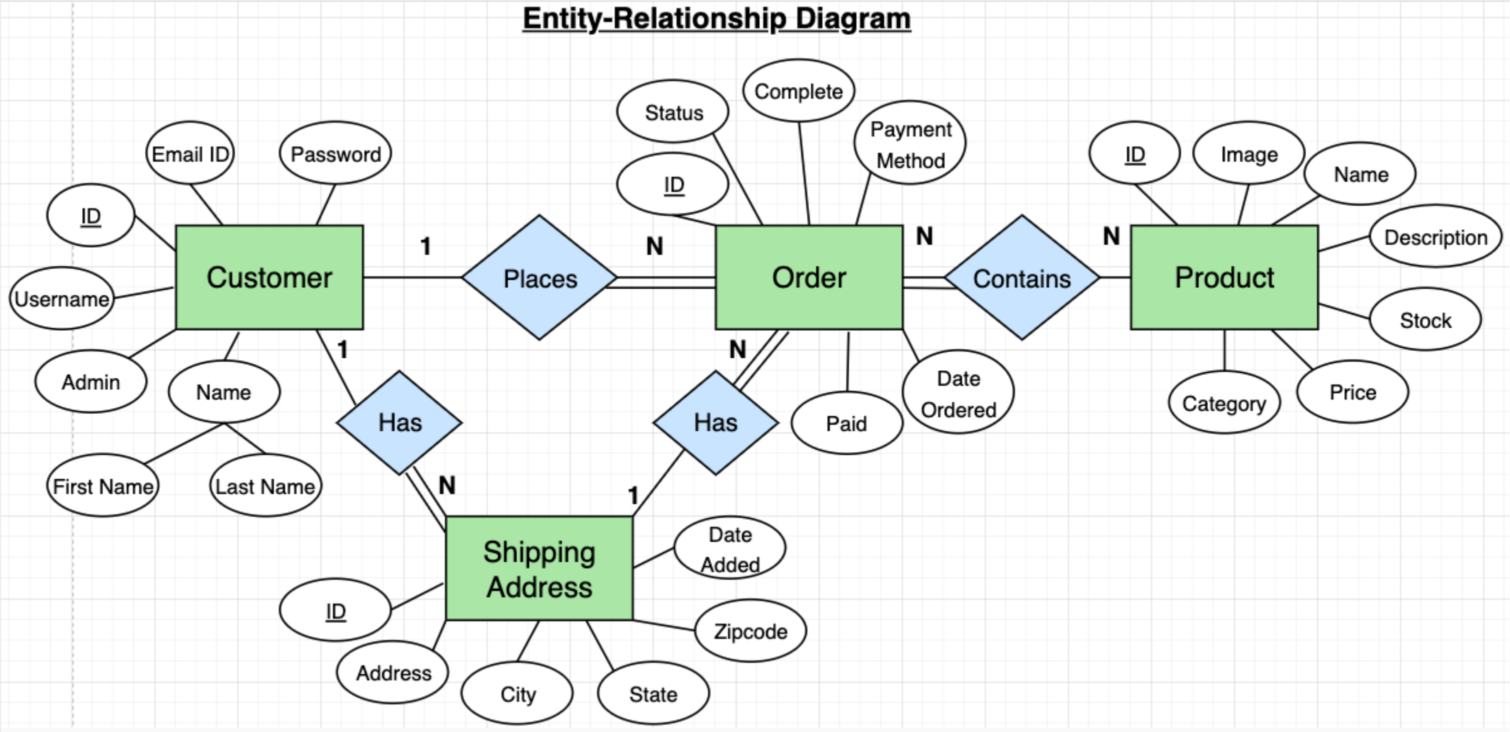
Typical Admin Navigation Process


Access Admin Website
Login using Admin credentials
Access Admin Homepage
View product inventory
View orders placed by customers
Add/Update product details
Delete products
Has order been successfully delivered?
Yes
Mark order as completed
No
Has order status changed?
Yes
Update Order Status

Databases

Entity-Relationship Diagram

All of the data of the e-commerce website will be stored and managed using a relational database. This database will have a conceptual structure as shown in the ERD below. Here, underlined attributes represent primary keys.



Through the proceeding sections, I will explain the functionality and importance of each table in the database and explain the significance of certain entity-relationships in regards to the database logical and physical efficiency.

Customer Table

In order for any new customer or administrator to create an account on the website, we will need to store some essential information. This information will be store stored in the **Customer table,** containing the following fields. Administrators and customer data is stored in the same table as most of the required information is identical, except for their access rights, which will be controlled using the **'Admin'** field mentioned below.

NOTE: The character lengths for this table's fields are predefined by Django.

Field Name	Data Type	Description	Constraints
ID	Integer (Primary Key)	The primary key used to create a unique value for each record.	

First Name	Varchar (150)	First name of the customer creating the account. Middle names can also be input here.	Must not include any numbers or symbols.
Last Name	Varchar (150)	The last name of the customer creating the account.	Must not include any numbers or symbols.
Email ID	Varchar (254)	The email address of the customer, to provide updates/reach out to them.	Must include an '@'.
Username	Varchar (Unique)	Username chosen by new customer.	Must be unique (not taken by a previous customer).
Password	Varchar (128)	Password of new customer, used to secure account.	Raw passwords will not be stored in the database. Instead, a hash of the password will be stored, to maintain security.
Admin	Boolean	Used to determine whether the new user is an administrator or a regular shopper.	Default = False. Will be pre-determined in backend, un-editable by users, and will play a vital role in defining user functionality.

Product Table

The Product table would store the information of the products available for sale on my client's website.

To determine what information would essentially need to be included and displayed to the user, I had a conversation with my client[3]. After the discussion, these are the data fields we finalized upon:

Field Name	Data Type	Description	Constraints/Comments
ID	Integer (Primary Key)	The primary key used to create a unique value for each record.	
Name	Varchar (200)	Name of the product.	
Description	Text	Product description and details.	Should not have a maximum length constraint.

[3] Refer to Appendix – "Criteria B: Determining Product Fields"

Category	Varchar (10)	The category to which the product belongs (the product type).	There will be predetermined category choices available to the admin when creating a new product (E.g., oils, mists etc.).
Image	Varchar (100)	The image of the product.	Stores the URL of the image, which is then loaded into the page.
Price	Signed Integer	The price of the product in Indian Rupees.	Number must be zero or positive.
Stock	Signed Integer	The available product inventory – quantity of a product available for customers to buy.	Number must be zero or positive.

ShippingAddress Table

To store the address of the customer where the order will be delivered, we will need to create a **ShippingAddress** table.

As it is possible for a customer to have multiple shipping addresses, I have created a separate model to store these addresses rather than storing them as a field in the Customer table to prevent the creation of multiple **redundant records** in the Customer table.

This table will need to have **1 Foreign Key field**:

1. **Customer –** this is as:
    a. 1 customer **can have** many shipping addresses
    b. But many customers **cannot have** identical shipping address (i.e., all typed characters are identical)

Thus, the field table for this table looks as shown below:

Field Name	Data Type	Description
ID	Integer (Primary Key)	The primary key used to create a unique value for each record.
Customer ID	Bigint (Foreign Key)	The ID of the customer who is adding the shipping address.
Address	Varchar (200)	The address (house no., apartment no., street etc.) where the order is to be delivered.
City	Varchar (200)	The city where the order is to be delivered.
State	Varchar (200)	The state where the order is to be delivered.

Zip code	Varchar (200)	The zip code where the order is to be delivered.
Date added	Datetime	The date & time when the shipping address is added. Automatically added when record is created.

Order Table

The Order table will represent the customer's "cart" while they browse the website, and will be stored as a complete customer order once the customer successfully places the order.

Every order record will reference 3 relational tables, as seen from the ERD given at the beginning:

1. **Customer table** – this must exist as:
    a. 1 customer **can place** many orders
    b. Many customers **cannot place** the same order

2. **ShippingAddress table** – this must exist as:
    a. Many orders **can contain** the same shipping address
    b. 1 order **cannot contain** many shipping addresses

3. **Many-to-Many Product relationship** – this must exist as:
    a. 1 order **can contain** many products
    b. Many orders **can contain** the same product

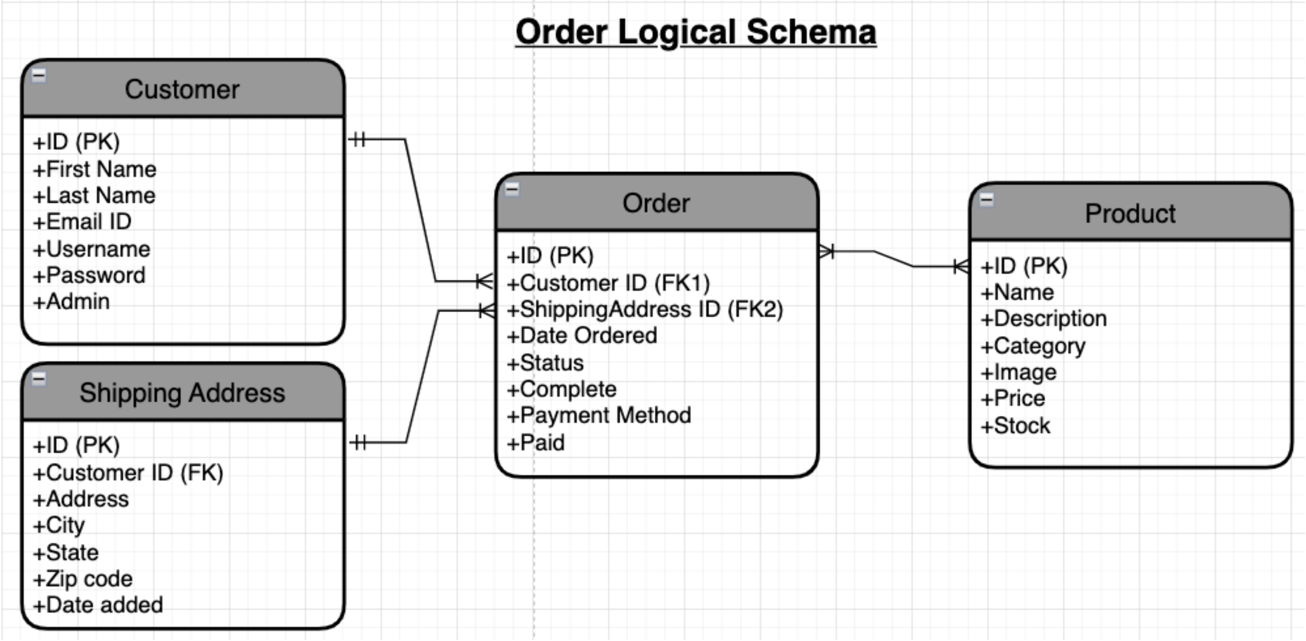
These entity relationships have been included to put the database into the **3rd Normal Form** and eliminate transitive dependencies between the mentioned tables and the Order table. Doing so reduces the number of redundant records stored, hence **reducing the physical storage space** occupied by the database while **increasing the efficiency of querying data.**

Thus, the data fields of the Order table will be as follows:

Field Name	Data Type	Description	Constraints/Comments
ID	Integer (Primary Key)	The primary key used to create a unique value for each record.	
Customer ID	Bigint (Foreign Key)	The Primary Key ID of the customer who is placing the order.	
Shipping Address ID	Bigint (Foreign Key)	The Primary Key ID of the shipping address record related to this order.	
Date Ordered	Datetime	The exact date & time when the order was placed.	Automatically added when record is created.

Status	Varchar (20)	The status of the order.	Default = “Received”. Will be selected from pre-determined options like “Received” and “Dispatched”, Will be editable by admin.
Complete	Boolean	Used to determine whether order has been successfully placed.	Default = False. Once order is processed and updated in the database, complete will be set to True.
Payment Method	Varchar (20)	The method of payment used by the customer.	Client mentioned in success criteria that various payment methods should be possible. This allows them to track the payment method used for each order.
Paid	Boolean	Used to determine whether payment is completed.	Default = False. As Cash on Delivery is a possible payment method, the payment would not be done at the time of order placement. Such cases must be reflected to administrator.

This is what the **logical schema centred around the Order table** specifically would look as follows, including all associated entity relationships:



However, you will notice here that the Order & Product table do not contain any direct relational fields between each other. This is as for **many-to-many relationships,** we must use a **linking table.**

OrderItem Table

To logically implement this many-to-many relationship, another table, called **OrderItem** will be created to serve as the **linking table.**

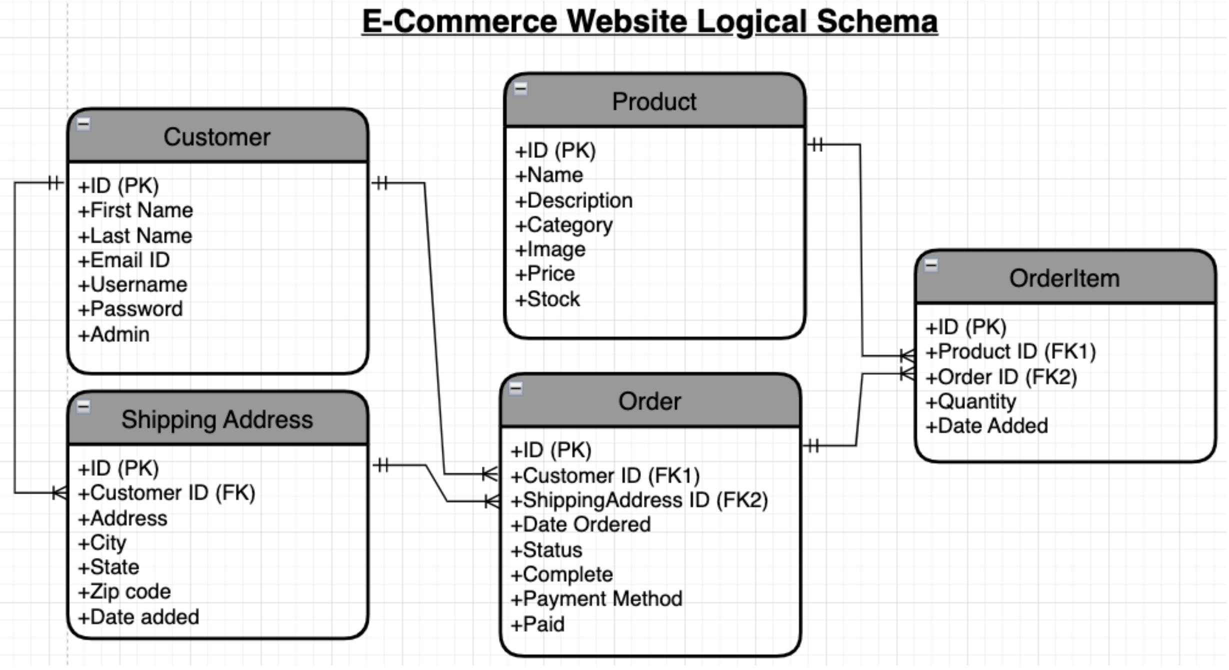
This table will independently record the products added to each customer's order. I.e., Thus, every time a new product is added to a customer's cart – a new record will be created in the OrderItem table.

This is what the fields of the OrderItem table will be:

Field Name	Data Type	Description	Constraints/Comments
ID	Integer (Primary Key)	The primary key used to create a unique value for each record.	
Product ID	Bigint (Foreign Key)	The ID of the product being added to an order.	
Order ID	Bigint (Foreign Key)	The ID of the order to which the product is being added.	
Quantity	Signed Integer	The quantity of the product.	Default quantity = 0. A customer can order multiple units of a single product, requiring a quantity field to track this. Can only store zero or positive values.
Date Added	Datetime	The date and time when the product was added to the order.	Automatically added when record is created.

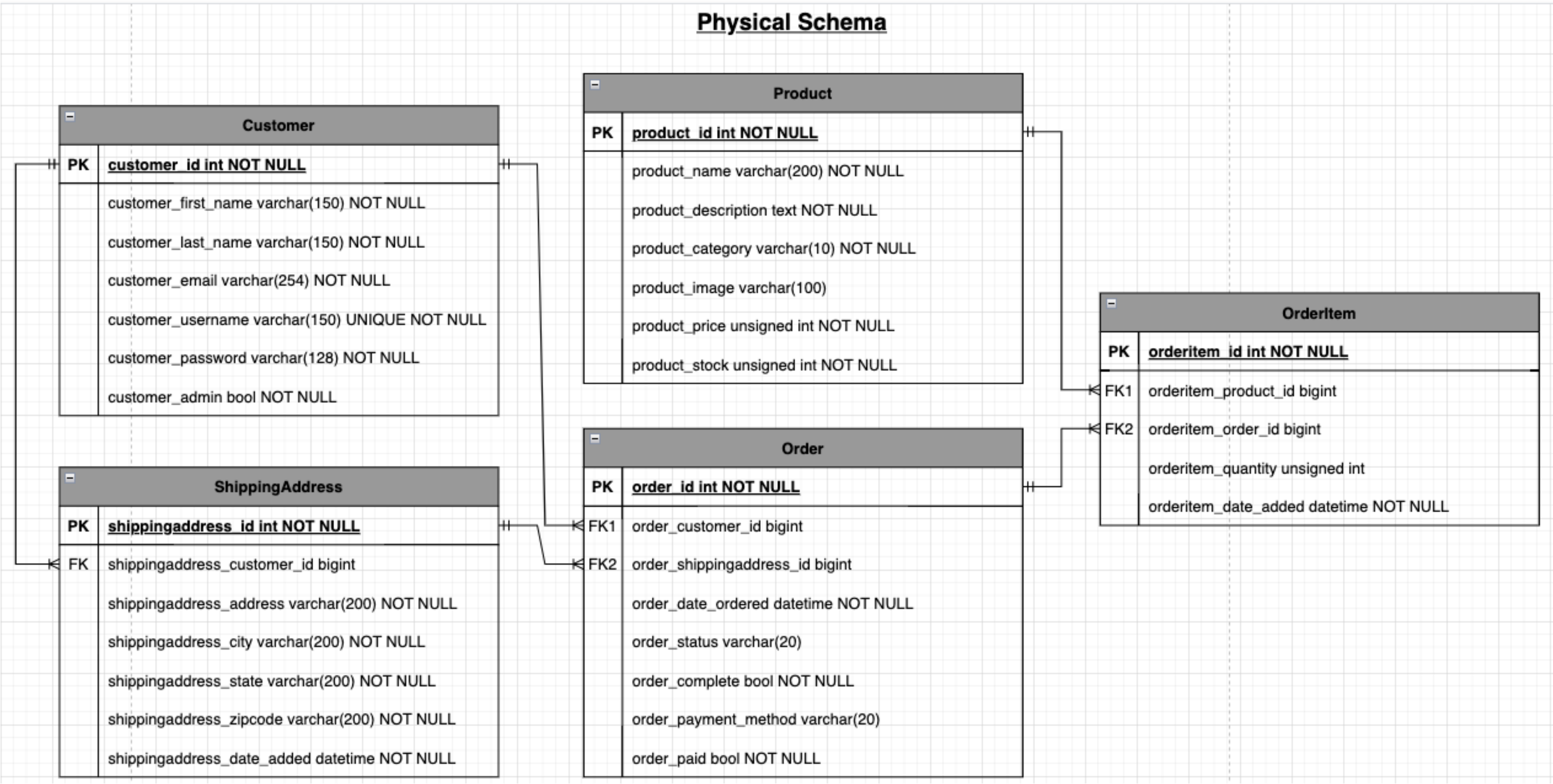
Logical Schema

Given all the tables, their attributes, and the relationships between them, the following logical schema can be created to represent the relational database.



Physical Schema of Website

Based on the created logical schema created and the attributes defined in the tables above, the following physical schema can be created to represent the relational database of the e-commerce website:



Physical Schema

Customer	
PK	customer_id int NOT NULL
	customer_first_name varchar(150) NOT NULL
	customer_last_name varchar(150) NOT NULL
	customer_email varchar(254) NOT NULL
	customer_username varchar(150) UNIQUE NOT NULL
	customer_password varchar(128) NOT NULL
	customer_admin bool NOT NULL

ShippingAddress	
PK	shippingaddress_id int NOT NULL
FK	shippingaddress_customer_id bigint
	shippingaddress_address varchar(200) NOT NULL
	shippingaddress_city varchar(200) NOT NULL
	shippingaddress_state varchar(200) NOT NULL
	shippingaddress_zipcode varchar(200) NOT NULL
	shippingaddress_date_added datetime NOT NULL

Product	
PK	product_id int NOT NULL
	product_name varchar(200) NOT NULL
	product_description text NOT NULL
	product_category varchar(10) NOT NULL
	product_image varchar(100)
	product_price unsigned int NOT NULL
	product_stock unsigned int NOT NULL

Order	
PK	order_id int NOT NULL
FK1	order_customer_id bigint
FK2	order_shippingaddress_id bigint
	order_date_ordered datetime NOT NULL
	order_status varchar(20)
	order_complete bool NOT NULL
	order_payment_method varchar(20)
	order_paid bool NOT NULL

OrderItem	
PK	orderitem_id int NOT NULL
FK1	orderitem_product_id bigint
FK2	orderitem_order_id bigint
	orderitem_quantity unsigned int
	orderitem_date_added datetime NOT NULL

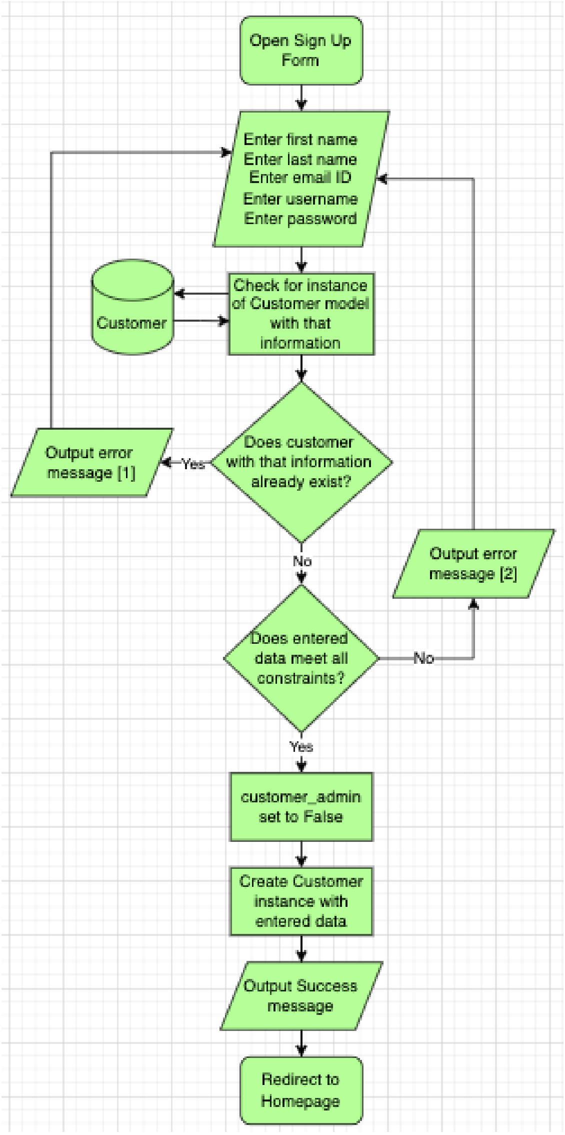
User-End Functionality

Signing up New Customers

For any new customer to create a new account on the website, a record with their details must be added to the **Customer table**. However, they will have **'admin' field set to False,** so that the **website's customers cannot access admin functionality.**

Reference for Flowchart:

- **Error message 1:** "An account with this information already exists."
- **Error message 2:** "There is ____ error in ____ field." (Blanks will be filled with the specific error and specific field)


Open Sign Up Form
Enter first name
Enter last name
Enter email ID
Enter username
Enter password
Customer
Check for instance of Customer model with that information
Does customer with that information already exist?
Yes
Output error message [1]
No
Does entered data meet all constraints?
No
Output error message [2]
Yes
customer_admin set to False
Create Customer instance with entered data
Output Success message
Redirect to Homepage

Pseudocode

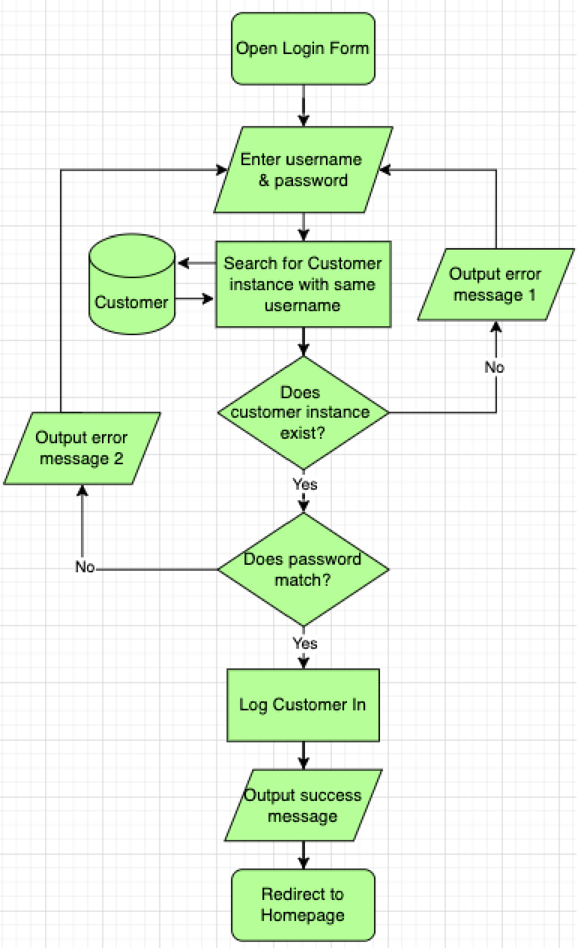
```
CUSTOMER = None

Loop while CUSTOMER not signed up AND User on signup page
    Read FIRST_NAME
    Read LAST_NAME
    Read EMAIL
    Read USERNAME
    Read PASSWORD

    CUSTOMER = Customer.get(username=USERNAME)

    If CUSTOMER == None then
        If constraints met then
            Customer.create(first_name = FIRST_NAME,
                            last_name = LAST_NAME,
                            email = EMAIL,
                            username = USERNAME,
                            password = hash(PASSWORD),
                            admin = False)
            Output "Account created."
            Redirect to homepage
        Else
            Output "There is ___ error in ___ field."
        End if

    Else
        Output "An account with this information already exists."
    End if

End loop
```

Logging in Existing Customers

However, if a customer already has an account (a record in the **Customer table**), only the **username** and **password** of the returning customer must be checked.

Pseudocode

```
Error Message 1 = “No account found with this username.”
Error Message 2 = “Incorrect password entered.”
CUSTOMER = None

Loop while CUSTOMER not logged in AND User on login page
        USERNAME = read “user_input”
        PASSWORD = read “user_input”

        CUSTOMER = Customer.get(username=USERNAME)

        If NOT CUSTOMER == None then
              If CUSTOMER.password = hash(PASSWORD) then
                      Log customer in
                      Output “Logged in.”
              Else
                      Output Error Message 2
              End if

        Else
              Output Error Message 1
        End if
End loop

Redirect to homepage
```



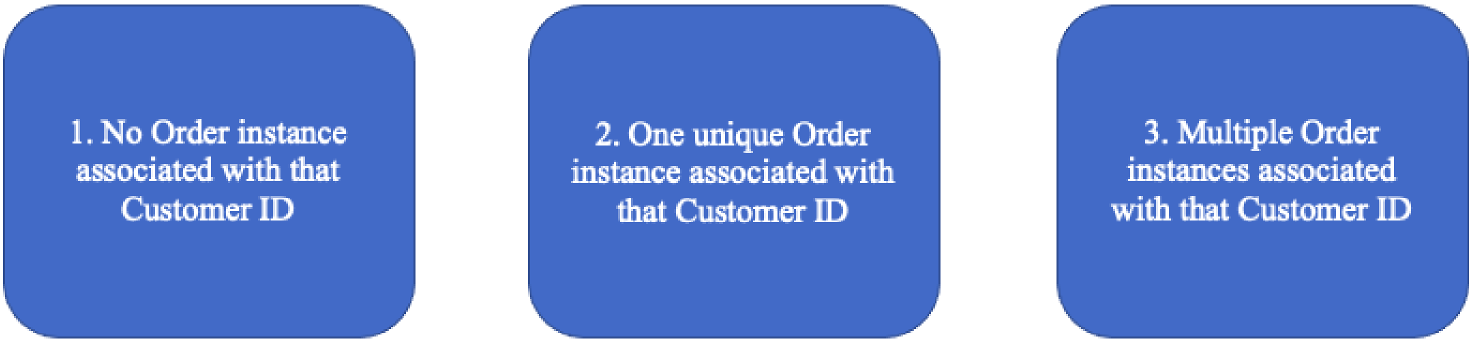
At this juncture, we must note that **signing up/logging in will be the first thing** that the customer must do to proceed on the website. Thus, **if they attempt any action** while not logged in (E.g., accessing cart, adding products etc.), they will be redirected to the sign up/log in page.

Creating/Retrieving Customer Cart

The **customer’s cart** is the place wherein they can add the products they wish to buy. This cart is simply a record **of the Order table** which is **associated with a specific logged in customer via a foreign key relationship.**

Thus, whenever a customer logs in and loads up the website, they should see a **cart (Order table record) associated with their ID,** to which they can add their products.

To do so, we would need to retrieve the order if an Order record already exists, or create one if it doesn't.



1. No Order instance associated with that Customer ID

2. One unique Order instance associated with that Customer ID

3. Multiple Order instances associated with that Customer ID

There are 3 possible scenarios when creating/retrieving an order:
The 3rd scenario is particularly unique. In this scenario, the customer has already successfully placed multiple orders in the past.

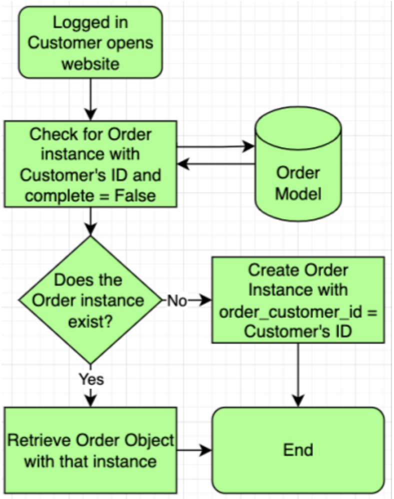
Thus, if we simply try to retrieve the Order record by checking the records with **order_customer_id = Customer_ID**, there will be **multiple order records** retrieved.

Hence, we need **another criterion for filtering.** This is where the **order_complete field** is used. With **default = False**, **order_complete** will be set to **True only** when an order has been **successfully placed.**

So, when we wish to check if a customer has an Order record associated with their ID, we would check for **order_customer_id = Customer_ID AND order_complete = False.**

<u>Pseudocode</u>
//First line shown for clarity. The customer would already be logged in, so the CUSTOMER variable would already have been set.

```
CUSTOMER = Customer.get(username = USERNAME)

ORDER = Order.get(customer_id = CUSTOMER_id, complete = False)

//Get function returns None if record doesn't exist

If ORDER == None then
        Order.create(customer_id = CUSTOMER_id)
End if
```


Logged in Customer opens website
Check for Order instance with Customer's ID and complete = False
Order Model
Does the Order instance exist?
No
Create Order Instance with order_customer_id = Customer's ID
Yes
Retrieve Order Object with that instance
End

Adding/Deleting/Updating Products to Cart

When a product is added to a customer's cart, a record in the **OrderItem table** is being created or updated. When a customer adds a product, one of 3 things can happen:

A new product is added to the cart

The quantity of a product in the cart is updated

A product is deleted from the cart (quantity of product in cart <= 0)

To **perform any of these 3 actions,** we must **use buttons.** These **buttons would have the following names & actions:**

- "Add" – add a new product/add 1 to the quantity of the product in the cart
- "Remove" – subtract 1 to the quantity of the product in the cart
- "Delete" – remove the product from the cart completely

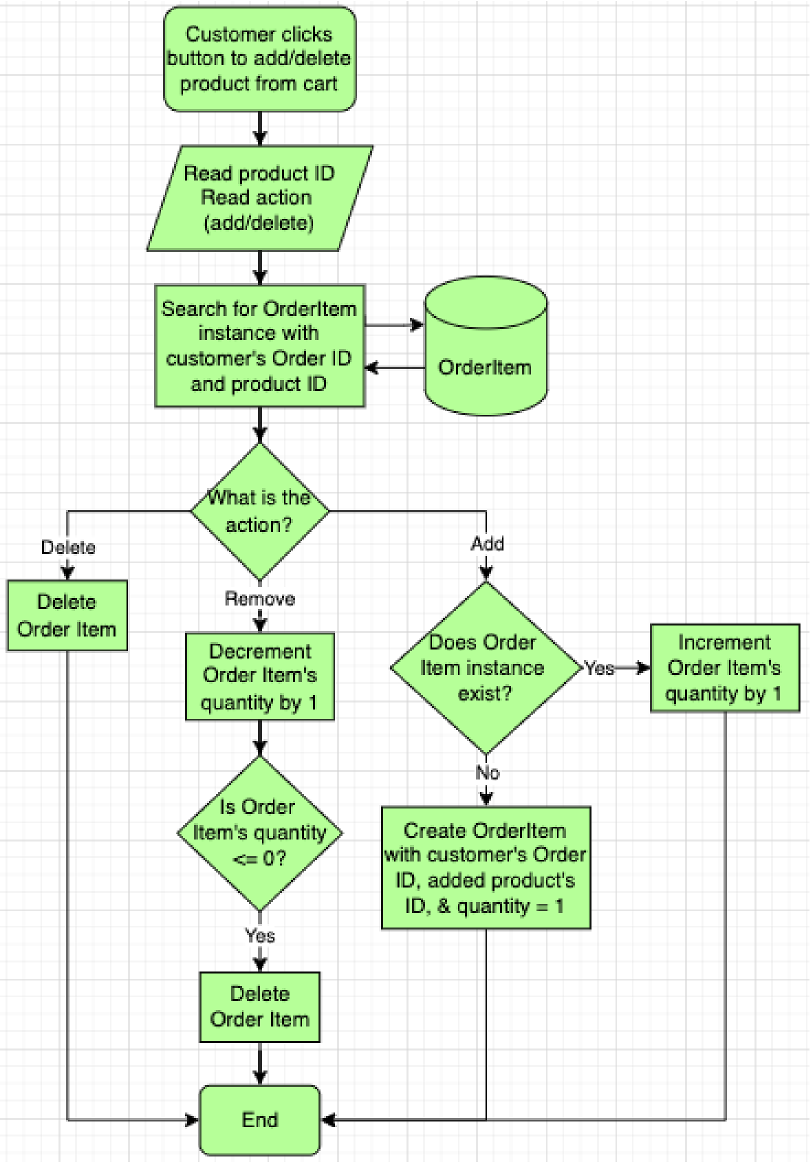
Here, it must be noted that the "**Remove" and "delete" buttons** will only be **clickable** for **products already in cart** (OrderItem records already exist)

The flowchart and pseudocode below showcase the **logic that will be undertaken** in **each of these scenarios.**

Pseudocode

```
//When button is clicked:
Read PRODUCT_id          //Primary Key ID associated with product

Read ACTION              //Action associated with button (i.e., Add, remove, or delete product)

//Lines below shown for clarity, Order & Customer records would've been retrieved by this point
CUSTOMER = Customer.get(username = USERNAME)

ORDER = Order.get(customer_id = CUSTOMER_id, complete = False)


ORDERITEM = OrderItem.get(order_id = ORDER_id, product_id = PRODUCT_id)

// "Remove" & "Delete buttons will only be visible for products already in cart
If ACTION == "Delete" then
            ORDERITEM.delete()

Else if ACTION == "Remove" then
    ORDERITEM_quantity = ORDERITEM_quantity – 1
```

```
        If ORDERITEM_quantity <= 0 then
                ORDERITEM.delete()
        End if

Else if ACTION == "Add" then
        If NOT ORDERITEM == None then
                ORDERITEM_quantity = ORDERITEM_quantity + 1

        Else
                OrderItem.create(order_id = ORDER_id,
                                 product_id = PRODUCT_id,
                                 quantity = 1)
        End if

End if
```


Customer clicks button to add/delete product from cart
Read product ID Read action (add/delete)
Search for OrderItem instance with customer's Order ID and product ID
OrderItem
What is the action?
Delete
Remove
Add
Delete Order Item
Decrement Order Item's quantity by 1
Does Order Item instance exist?
Yes
Increment Order Item's quantity by 1
No
Is Order Item's quantity <= 0?
Create OrderItem with customer's Order ID, added product's ID, & quantity = 1
Yes
Delete Order Item
End

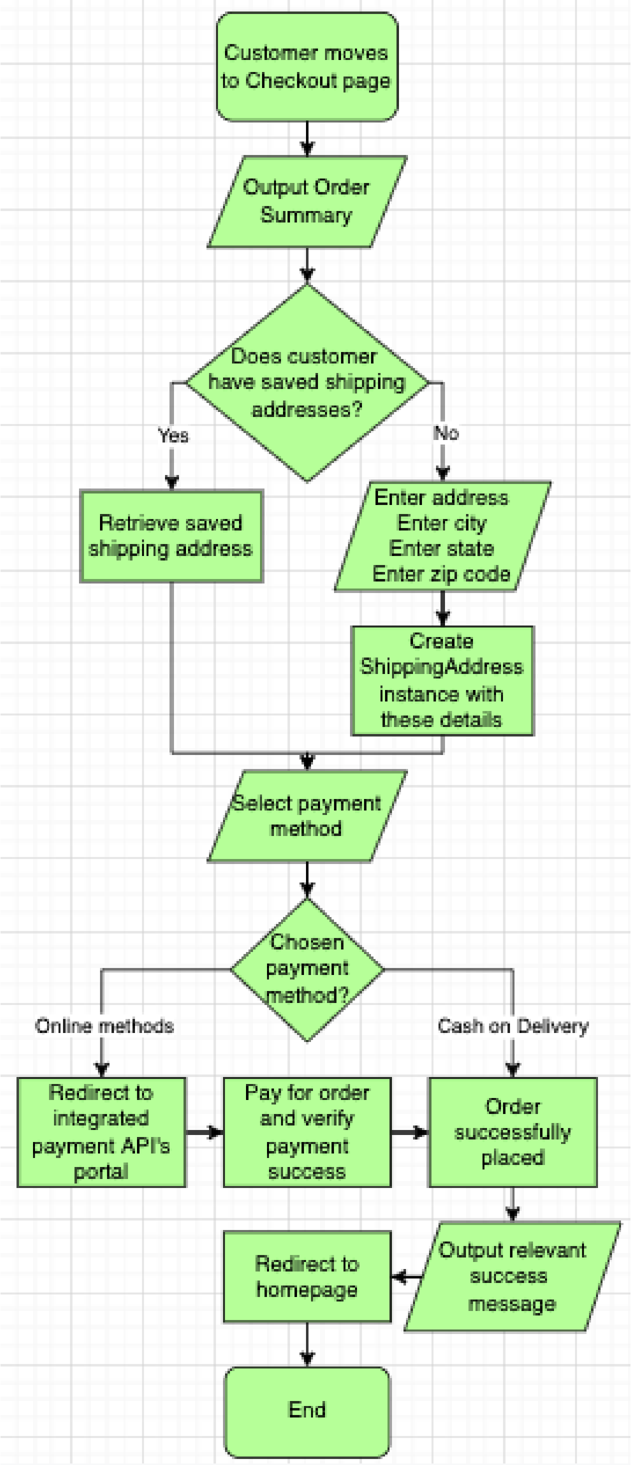
Checkout Flow

When the customer moves to the checkout stage in order to place the order, there is a typical flow they will follow. This will involve:

- Entering the shipping address
- Choosing a payment method
- Paying for the order

This checkout flow will be as shown in the flowchart below:

NOTE: The logic for both online methods (UPI or debit/credit cards) & cash on delivery has been included as the client had explicitly mentioned in the success criteria that customers should be able to pay through **either method.**



Customer moves to Checkout page
Output Order Summary
Does customer have saved shipping addresses?
Yes
No
Retrieve saved shipping address
Enter address
Enter city
Enter state
Enter zip code
Create ShippingAddress instance with these details
Select payment method
Chosen payment method?
Online methods
Cash on Delivery
Redirect to integrated payment API's portal
Pay for order and verify payment success
Order successfully placed
Output relevant success message
Redirect to homepage
End

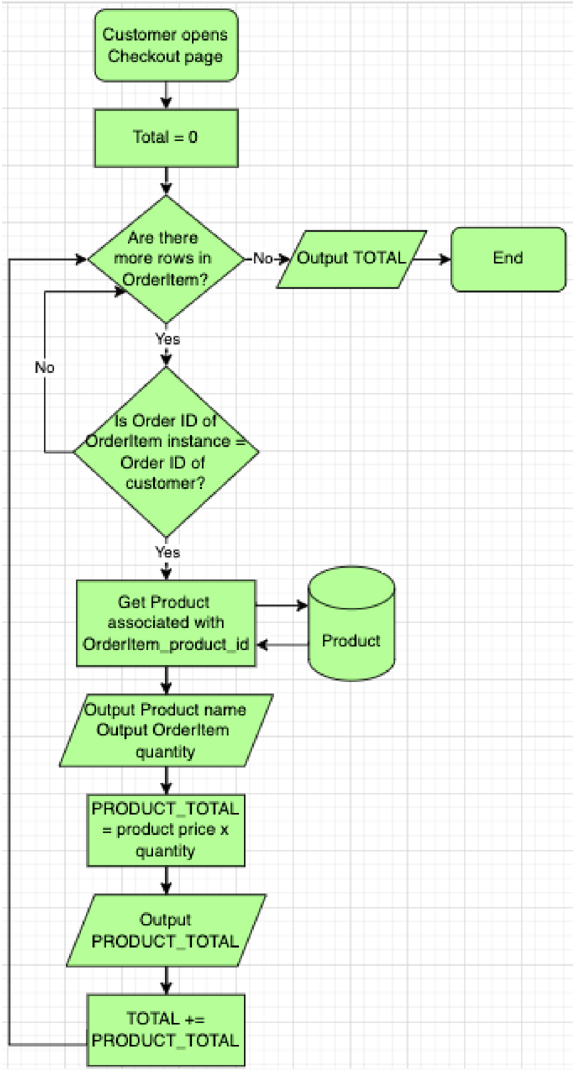
Checkout – Presenting Order Summary

When the customer moves to the checkout page, they should be able to see a summary of the order they are placing, including:

- The products they are ordering
- The quantity of each product
- The total price of each product (quantity x unit price)
- The total price of the order

To display this information, we would have to retrieve **all the OrderItem records associated with that order,** and extract the required information from them.

The logic and flowchart for this would be as follows:


Customer opens Checkout page
Total = 0
Are there more rows in OrderItem?
No
Output TOTAL
End
Yes
Is Order ID of OrderItem instance = Order ID of customer?
No
Yes
Get Product associated with OrderItem_product_id
Product
Output Product name Output OrderItem quantity
PRODUCT_TOTAL = product price x quantity
Output PRODUCT_TOTAL
TOTAL += PRODUCT_TOTAL

Pseudocode

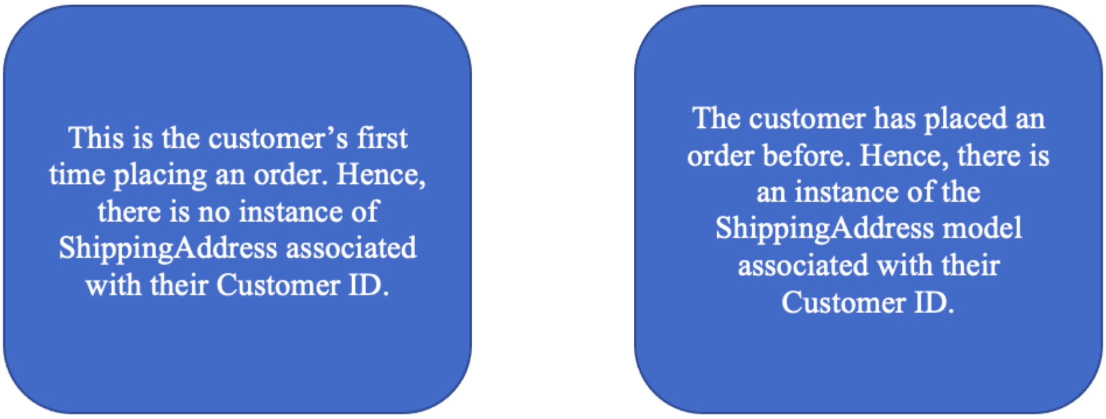
```
//First 2 lines shown for clarity, Order & Customer records
would've been retrieved by this point

CUSTOMER = Customer.get(username = USERNAME)
ORDER = Order.get(customer_id = CUSTOMER_id, complete =
False)

TOTAL = 0

//Looping through all OrderItemrecords associated with ORDER
Loop for ITEM in OrderItem.get(order_id = ORDER_id)
        PRODUCT = Product.get(id = ORDERITEM_product_id)
        Output PRODUCT_name
        Output ORDERITEM_quantity

        PRODUCT_TOTAL = PRODUCT_price * ORDERITEM_quantity
        Output PRODUCT TOTAL   //Total price for each product

        TOTAL = TOTAL + PRODUCT_TOTAL
End loop

Output TOTAL          //Total price of order
```

Checkout – Shipping Address

When a customer opens the checkout page, they will need to enter their shipping details. This is important as my client needs to know the address to which the products must be delivered in order to be able to deliver them.

To fill in shipping details, the customer will need to fill up a form with the details required by the ShippingAddress table.

However, as the **ShippingAddress table has a Foreign Key relationship with the Customer table**, there are two possible scenarios:

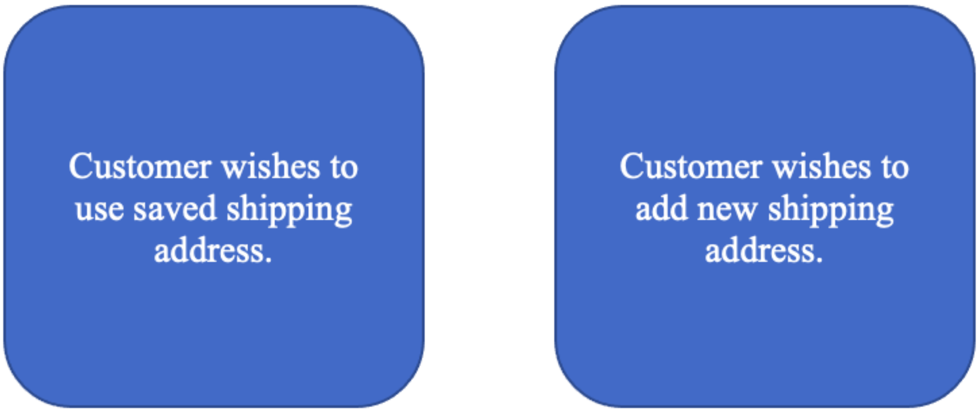


This is the customer's first time placing an order. Hence, there is no instance of ShippingAddress associated with their Customer ID.

The customer has placed an order before. Hence, there is an instance of the ShippingAddress model associated with their Customer ID.

Thus, the logic of the shipping details will have to be such that it **first checks for existing shipping address records associated with the Customer's ID,** and if those do not exist, then requiring the customer to fill out the form.

However, we must also consider that **<u>if a customer already has saved Shipping addresses</u> (previous shipping address records)**; there are still 2 possibilities:



Customer wishes to use saved shipping address.

Customer wishes to add new shipping address.

In the first case, we may simply **retrieve the existing ShippingAddress record** that the customer chooses. However, if the customer wishes to **add a new shipping address,** then we must show the **shipping details form.**

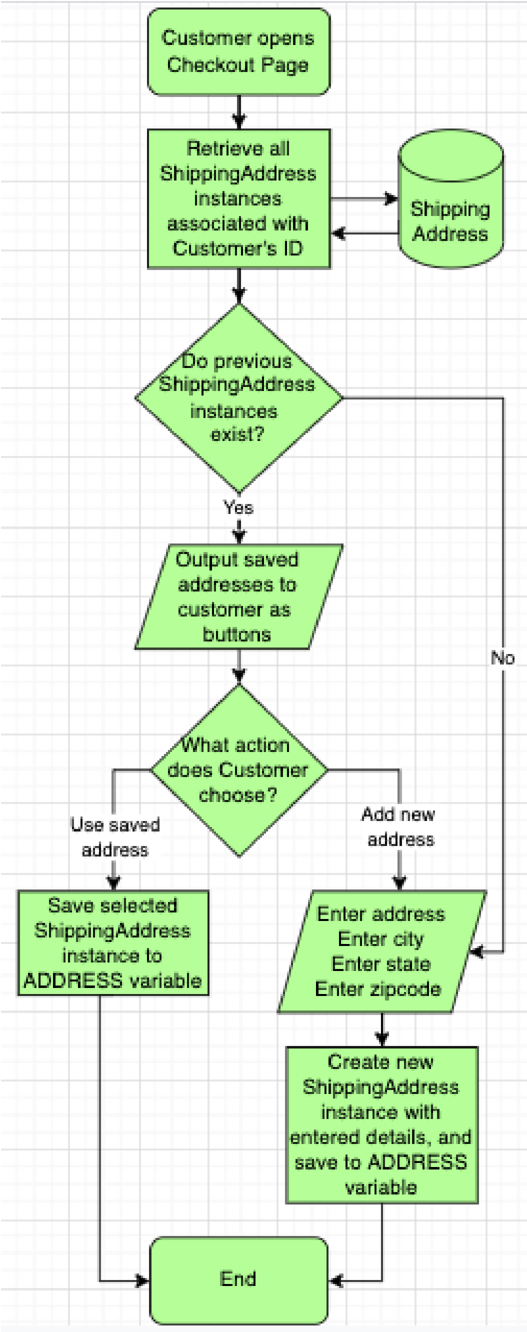
Thus, we must also **add buttons for each scenario** (E.g., use saved address or add new address); which the customer must click to determine which action they wish to take.

Thus, the pseudocode and flowchart for this process would be as follows:

NOTE: Pseudocode has been split into different sections, to provide clarity as to how data and inputs will move between the frontend and backend for a final shipping address to be selected.

Pseudocode 1 (Backend): Retrieving Saved Shipping Addresses

```
//Line below shown for clarity. The customer would already be logged in, so the CUSTOMER variable would already have been set.

CUSTOMER = Customer.get(username = USERNAME)

ADDRESSES COLLECTION
//To store possible multiple shipping addresses of customer

Loop for I in ShippingAddress.get(customer_id = CUSTOMER_id)
    ADDRESSES.addItem(I)
End loop
```


Customer opens Checkout Page
Retrieve all ShippingAddress instances associated with Customer's ID
Shipping Address
Do previous ShippingAddress instances exist?
Yes
No
Output saved addresses to customer as buttons
What action does Customer choose?
Use saved address
Add new address
Save selected ShippingAddress instance to ADDRESS variable
Enter address
Enter city
Enter state
Enter zipcode
Create new ShippingAddress instance with entered details, and save to ADDRESS variable
End

Pseudocode 2 (Frontend): Presenting Saved Shipping Addresses (if any) to customer

```
//Shipping addresses presented as clickable buttons with their own unique IDs

X = 0

If NOT ADDRESSES.isEmpty() then
    Loop while ADDRESSES.hasNext()
        DATA = ADDRESSES.getNext()
        BUTTON = button.create(ID = X, data = DATA)
        Output(BUTTON)
        X = X + 1
    End loop
End if
```

Pseudocode 3 (Backend): Creating New Shipping Address or Selecting Existing One

```
//Method used to prevent rewriting code in nested 'else if' and main 'else' blocks

Method addAddress(ID)
    Read HOME_ADDRESS
    Read CITY
    Read STATE
    Read ZIPCODE

    ADDRESS = ShippingAddress.create(customer_id = ID,
                                     address = HOME_ADDRESS,
                                     city = CITY,
                                     state = STATE,
                                     zipcode = ZIPCODE)
    Return ADDRESS
End addAddress

If NOT ADDRESSES.isEmpty() then
    If clickOn("Saved address" button) then
        Read button_id          //Read ID of clicked button
        ADDRESS = data for button with ID = button_id

    Else if clickOn("Add new address" button) then
        ADDRESS = addAddress(CUSTOMER_id)

    End if

Else
    ADDRESS = addAddress(CUSTOMER_id)

End if
```

Checkout – Payment Integration

The next step in the process would be to select a payment method. As mentioned in the scope, these are the payment methods that will be available on the website:

- Cash on Delivery
- Credit/Debit cards
- UPIs like Paytm and Google Pay

For Cash on Delivery, no transaction would have to be made at the time of placing the order; and the order would instead automatically be placed successfully **(this is discussed in detail later)**. However, if **online methods** of payment such as **credit/debit cards** or **UPIs** are used, then we would need to handle **online transactions.**

To do so, I would need to **integrate an application programming interface (API) for payments** into my website. This would route the payment to an external payment gateway, which would handle the process of **connecting the backend with the chosen frontend bank/UPI,** allowing for secure payment.

For this, I have chosen to use the **Razorpay API.** This is as it[4,5]:

- Allows payment via all the methods the client desires

[4] Mehta, Rohit. "Top 5 Online Payment Gateway in India." *Times of India Blog*, 18 Sept. 2021, https://timesofindia.indiatimes.com/blogs/digital-mehta/top-5-online-payment-gateway-in-india/. Accessed 28 Feb 2022.

[5] Razorpay. "Razorpay Payment Gateway." *Razorpay*, https://razorpay.com/payment-gateway/. Accessed 28 Feb 2022.

- Has easy integration via JavaScript with minimal steps
- Has a strong testing system in place, allowing for testing to happen in all scenarios.
- Provides strong end-to-end security
- Is used by many renowned Indian companies such as UrbanCompany and Grofers
- Takes only 2-3% per transaction as its commission; compared to other companies, which can take 3% plus some additional taxes/fixed fees.

For the payment integration, I will be following the steps mentioned in the Razorpay API documentation[6].

This documentation describes the necessary steps and essential logic that must be included in your code to integrate and verify payments.

The gist of the payment integration process is described below:

1. **<u>Create a Razorpay Order</u>**

This is an object created for the Razorpay API on the website's server. It takes the following information:

- Payment amount in paise (i.e., Rs. 100 is represented as 10,000 paise) –**value can be taken from the TOTAL variable defined in 'Checkout – Presenting Order Summary'**

- Currency of transaction (i.e., INR)

This object is created for Razorpay's API, as it is required to be sent alongside the payment request.

We must store the ID of this order object in our website server in the variable RAZORPAY_ORDER_ID (needed to verification later).

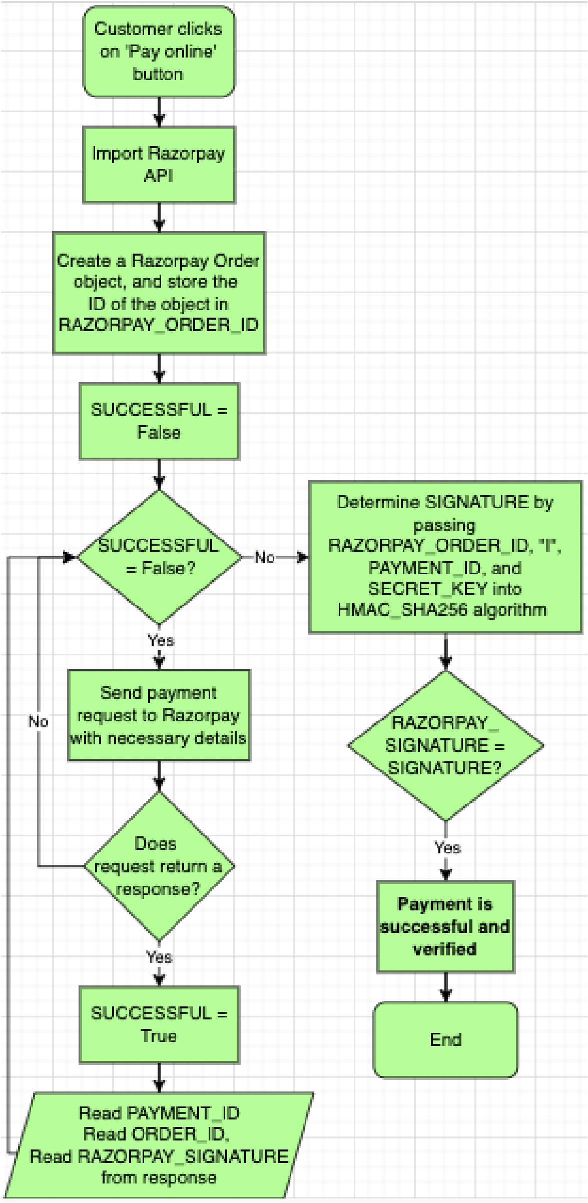
2. **<u>Send a Payment Request</u>**

When the user clicks a button to pay with Razorpay, **a payment request** must be sent to the **Razorpay server.** This request will include:

- The website server's public key **(key to ensure payment request is generated by website and not fraudulent source)**
- Amount in paise
- Currency (i.e., INR)
- Company name
- String ID of the Razorpay order object – RAZORPAY_ORDER_ID

[6] "Build Integration." *Razorpay Docs*, Razorpay, https://razorpay.com/docs/payments/payment-gateway/web-integration/standard/build-integration/. Accessed 28 Feb 2022.

If the payment fails, **Razorpay will inform the customer,** and **allow the customer to retry payment.**


Customer clicks on 'Pay online' button
Import Razorpay API
Create a Razorpay Order object, and store the ID of the object in RAZORPAY_ORDER_ID
SUCCESSFUL = False
SUCCESSFUL = False?
No
Determine SIGNATURE by passing RAZORPAY_ORDER_ID, "|", PAYMENT_ID, and SECRET_KEY into HMAC_SHA256 algorithm
Yes
Send payment request to Razorpay with necessary details
No
RAZORPAY_ SIGNATURE = SIGNATURE?
Yes
Does request return a response?
Payment is successful and verified
Yes
SUCCESSFUL = True
End
Read PAYMENT_ID Read ORDER_ID, Read RAZORPAY_SIGNATURE from response

3. <u>Verifying Payment</u>

If the **payment is successful,** the **Razorpay server** returns the following information to the website's server:

- A string with a unique payment ID – **RAZORPAY_PAYMENT_ID**
- A string ID of the Razorpay order object
- A string with a unique signature – **RAZORPAY_SIGNATURE**

To verify that the payment is credible and not fraudulent, we would need to **verify it.** To do so, the **following process will be conducted:**

A **hashed string signature** will then be produced using the **HMAC_SHA256 algorithm,** which will take the inputs of:

- RAZORPAY_ORDER_ID
- RAZORPAY_PAYMENT_ID
- SECRET_KEY – stored in the server beforehand
- An additional "|" character

If the **hashed string signature** generated by the website's server matches the string signature returned by Razorpay (RAZORPAY_SIGNATURE), **then the payment is considered verified.**

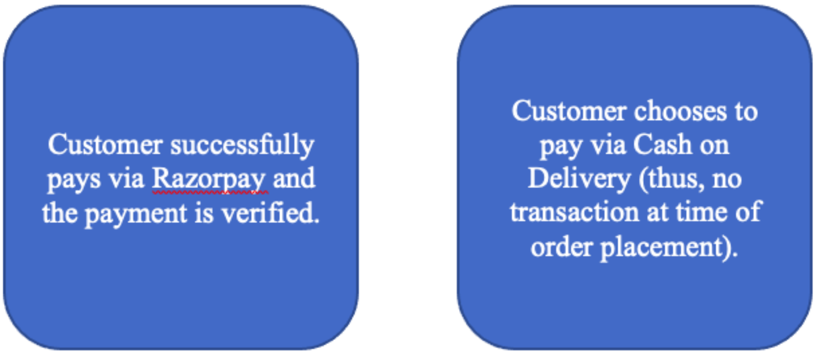
Thus, the following algorithm will be run to pay using online methods:

<u>Pseudocode</u>

```
Import Razorpay      //Module of Razorpay API

If clickOn("Pay online" button) then
    //Create a Razorpay Order
    RAZORPAY_ORDER = Razorpay.order.create(amount = TOTAL * 100, currency = "INR")
    RAZORPAY_ORDER_ID = RAZORPAY_ORDER_id


    //Send a Payment Request
    PUBLIC_KEY          //Will be stored in website server when integrating API
    SUCCESSFUL = False
    Loop while NOT SUCCESSFUL
        Razorpay.sendRequest(key = PUBLIC_KEY,
                             amount = TOTAL * 100,      //Amount stored in paise
                             currency = “INR”,
                             company_name = “KareKraft”,
                             order_id = RAZORPAY_ORDER_ID)

        If request returns a response then
            SUCCESSFUL = True

            //Read in from the information returned by Razorpay to the website’s server
            Read RAZORPAY_PAYMENT_ID
            Read RAZORPAY_ORDER_ID
            Read RAZORPAY_SIGNATURE
        End if
    End loop


    //Verify Payment
    SECRET_KEY        //Will be stored in website server when integrating API

    SIGNATURE = hmac_sha256(RAZORPAY_ORDER_ID + “|” +
                             RAZORPAY_PAYMENT_ID, SECRET_KEY)

    If SIGNATURE == RAZORPAY_SIGNATURE then
        Payment is successful and verified (code carried forward in next section)
    End if

End if
```

Checkout – Post Payment Processing

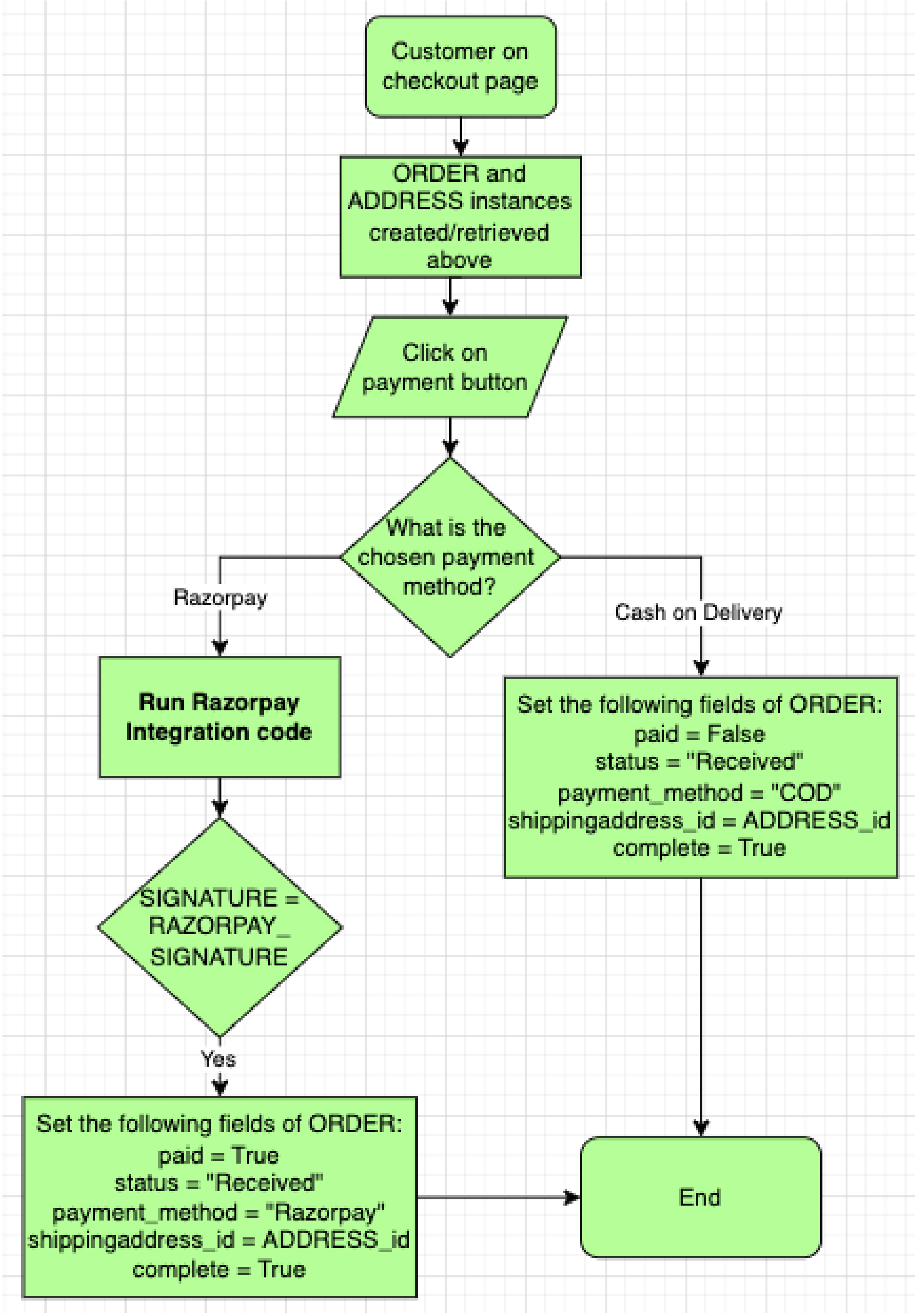
An order can be considered 'completed' and a cart can be closed when:



Customer successfully pays via Razorpay and the payment is verified.

Customer chooses to pay via Cash on Delivery (thus, no transaction at time of order placement).

Thus, the following logic will be executed when we move to the payment stage:


Customer on checkout page
ORDER and ADDRESS instances created/retrieved above
Click on payment button
What is the chosen payment method?
Razorpay
Cash on Delivery
Run Razorpay Integration code
SIGNATURE = RAZORPAY_ SIGNATURE
Yes
Set the following fields of ORDER: paid = True status = "Received" payment_method = "Razorpay" shippingaddress_id = ADDRESS_id complete = True
Set the following fields of ORDER: paid = False status = "Received" payment_method = "COD" shippingaddress_id = ADDRESS_id complete = True
End

Pseudocode

//Code carries on from successful payment verification
//First 2 lines shown for clarity, Order & Customer records would already exist.

```
CUSTOMER = Customer.get(username = USERNAME)
ORDER = Order.get(customer_id = CUSTOMER_id, complete = False)

ADDRESS   //ShippingAddress record created in ‘Checkout – Shipping Address’

If clickOn(“Pay online” button) then
        Run Razorpay integration logic mentioned in section above

        If SIGNATURE == RAZORPAY_SIGNATURE then
                ORDER_paid = True                  //Order paid for online
                ORDER_status = “Received”          //Default value

                ORDER_payment_method = “Razorpay”
                ORDER_shippingaddress_id = ADDRESS_id
                ORDER_complete = True
        End if


Else if clickOn(“Cash on Delivery” button) then
        ORDER_paid = False                 //Payment pending for order
        ORDER_status = “Received”          //Default value

        ORDER_payment_method = “Cash on Delivery”
        ORDER_shippingaddress_id = ADDRESS_id
        ORDER_complete = True
End if
```

Admin-End Functionality

Signing up New Admin Accounts

An administrator account is just a **record of the Customer table** with **Customer_admin = True.** This **field** makes the **customer into an admin,** enabling them **access to the administrator page & functionality.**

To create a new admin account, **an existing administrator would have to enter the details of the administrator to be added.** This would likely be managed as a panel under the existing **administrator's page.**

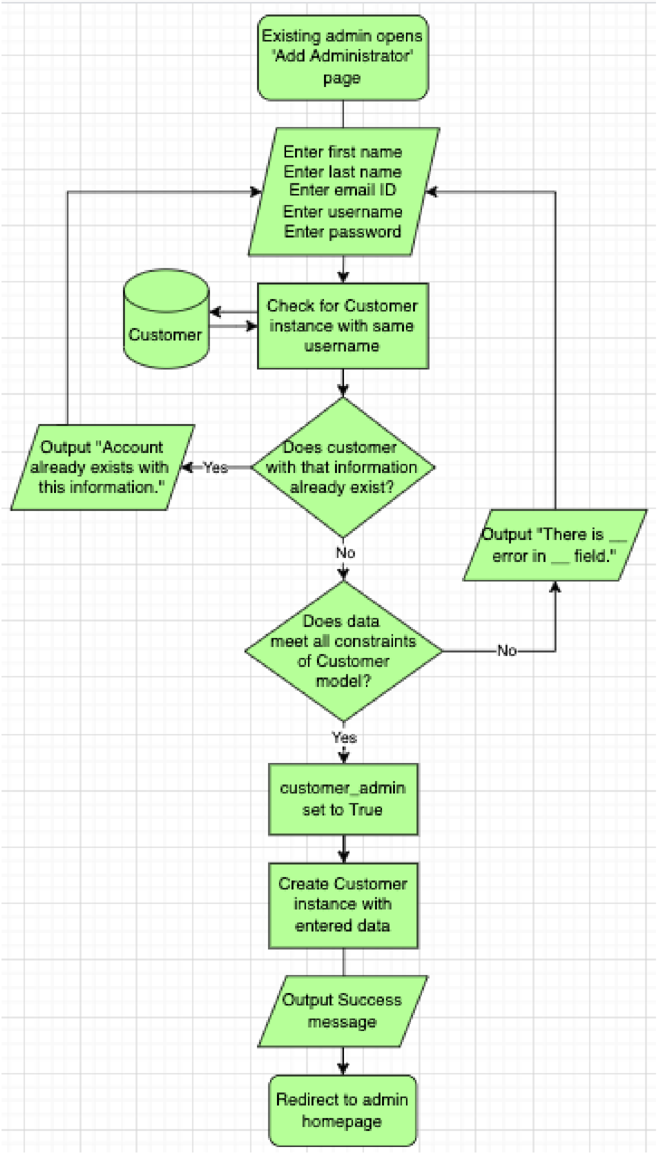
This needs to be done to ensure that **only existing administrators (E.g., my client)** can **permit new individuals to make administrative changes** to the website. If there were a **central signup page**, there would be **no control** over this.

For a new administrator, the following data would have to be entered:

- First name
- Last Name
- Email ID
- Username
- Password

All these **fields** would have to be **manually entered** by an **existing administrator** from the **page** from which they can **add administrators**. When the **new admin** is added, **customer_admin = True** will be **automatically set in backend.**

The process for **<u>logging in administrators</u>** would be the same as that for regular customers, thus, the logic has not been repeated[7].


Existing admin opens 'Add Administrator' page
Enter first name
Enter last name
Enter email ID
Enter username
Enter password
Customer
Check for Customer instance with same username
Output "Account already exists with this information."
Yes
Does customer with that information already exist?
No
Output "There is __ error in __ field."
Does data meet all constraints of Customer model?
No
Yes
customer_admin set to True
Create Customer instance with entered data
Output Success message
Redirect to admin homepage

Pseudocode for Sign Up

```
//Assuming admin on 'Add Administrator' page

ADMIN = None

Loop while ADMIN not added AND existing admin on 'add administrator' page
    //Existing admin enters details for new administrator
    Read FIRST_NAME
```

[7] Refer to 'Logging in Existing Customers' of this document for the flowchart and pseudocode for this process.

```
        Read LAST_NAME
        Read EMAIL
        Read USERNAME
        Read PASSWORD

        //Usernames are unique, thus can be the only parameter that is searched
        ADMIN = Customer.get(username=USERNAME)

        If ADMIN == None then
                If constraints met then
                        Customer.create(first_name = FIRST_NAME,
                                        last_name = LAST_NAME,
                                        email = EMAIL,
                                        username = USERNAME,
                                        password = hash(PASSWORD),
                                        admin = True)

                        Output "Account created."
                        Redirect to admin homepage
                Else
                        Output "There is ___ error in ___ field."
                End if

        Else
                Output "An administrator with this information already exists."
        End if

End loop
```

Adding Products

One key functionality of website administrators will be the ability to **add/delete products** available for sale on the e-commerce store, via the **administrator site.**

To do so, they will have to create a **new record** in the **Product table.**

The flowchart for adding products is attached alongside, where:

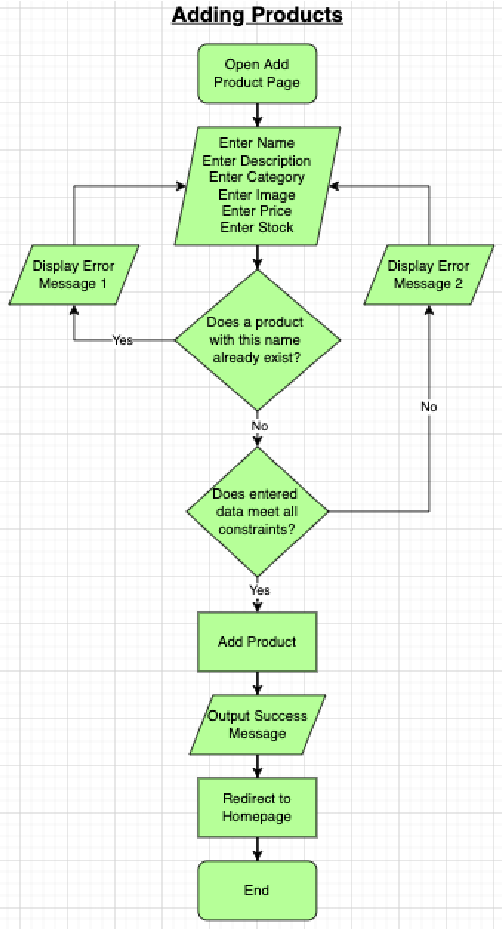
Error message 1: "A product with this name already exists."

Error Message 2: "There is ____ error in ____ field."

- Blanks will be filled with the specific error and specific field

Pseudocode

```
PRODUCT = None

Loop while PRODUCT not added AND admin on Products page
    Read NAME
    Read DESC
    Read CATEGORY
    Read IMAGE
    Read PRICE
    Read STOCK

    PRODUCT = Product.get(name = NAME)

    If PRODUCT == None then
        If constraints met then
            Product.create(name = NAME,
                           description = DESC,
                           category = CATEGORY,
                           image = IMAGE,
                           price = PRICE,
                           stock = STOCK)
            Output "Product Added."
            Redirect to homepage
        Else
            Output "There is ____ error in ____ field."
        End if
    Else
        Output "A product with this name already exists."
    End if
End loop
```



Deleting Products

To delete products, the admin will only have to confirm whether they want to delete the product. If they **confirm,** the specific **record in the Product table will be deleted.**

Pseudocode for deleting products:

Pseudocode

```
If clickOn(Delete button) then
    Output "Are you sure you want to delete the product?"
    Read BUTTON_INPUT
    If BUTTON_INPUT == "Yes" then
        Delete product
    End if
End if
```

Viewing All Orders

If the **administrator** wishes to be able to **view all orders placed** on the e-commerce store. All these orders would be presented as **buttons,** so that the **admin** can **click on any order** & **view all its details.**

Under the **order preview** the admin should be able see in the list of all orders would include:

- Order ID
- Customer ID
- Date Ordered
- Order status
- Payment method
- Paid

This is as **Order & Customer IDs are essential in identifying orders.**

Furthermore, **date ordered & 'paid' allow admin to determine priority of orders,** as those placed earlier or with paid = False would be of higher priority to complete.

To do so, **the logic will be as follows:**

Pseudocode

```
//Administrator opens 'View Orders' page

DATA ARRAY        //Array of length 6 to hold the data of each order preview

For ORDER in Order table
        DATA = {ORDER_id,
                    ORDER_customer_id,
                    ORDER_date_ordered,
                    ORDER_status,
                    ORDER_payment_method,
                    ORDER_paid}

        ORDER_BUTTON = button.create(ID = ORDER_id, data = DATA)
        Output ORDER_BUTTON
End loop
```

Viewing A Specific Order

Then, if the **admin clicks** on a **specific order button,** the **details of that order** would have to be presented. These **details** would **include the:**

- Order ID

- Name and email ID of the customer who has placed the order, retrieved from the Customer object associated with the order

- The shipping address to which the order has to be delivered, retrieved from the ShippingAddress object associated with the order

- Date ordered

- List of products ordered, which would be retrieved from the OrderItem record associated with that Order ID. The list would include:
    - Name of products, which would be retrieved from the Product record linked to each OrderItem record

    - Quantity of each product ordered

- The total price of the order (retrieved from TOTAL variable calculated during checkout)

- Complete Boolean value

- Order status

- Payment method

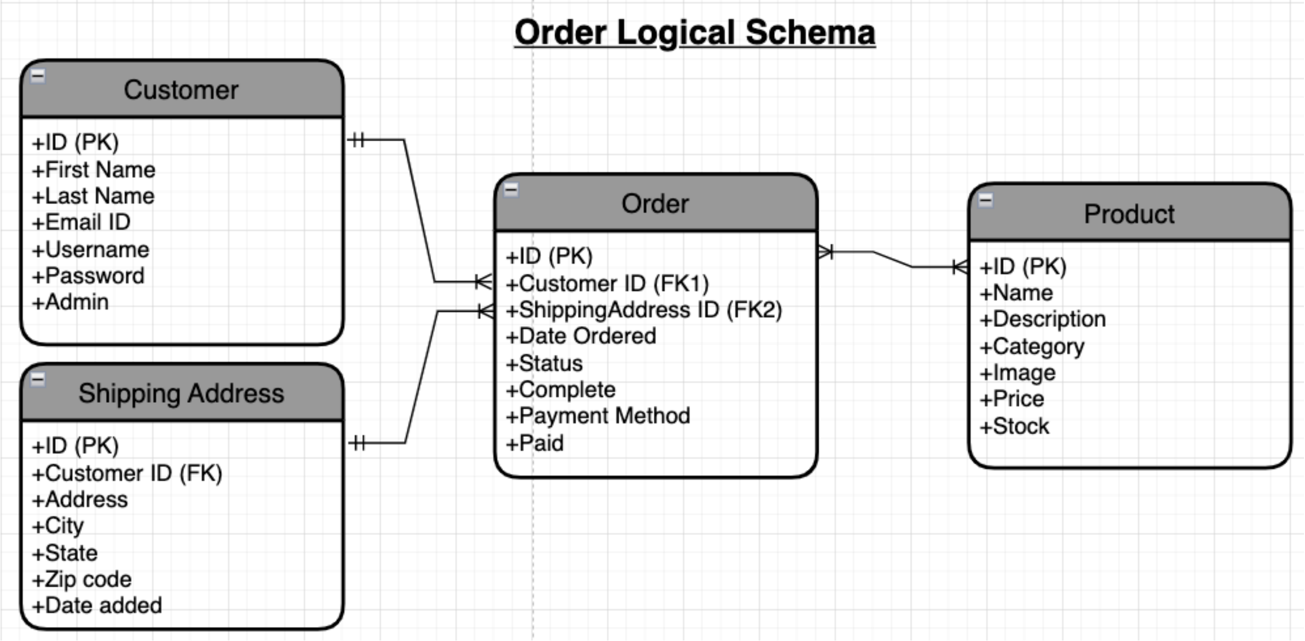
- Paid Boolean value

This would be done using the **pseudocode shown below.** The **Order table's logical schema** has also been **attached for reference below,** to provide **more clarity** as to **the relationships via which data is being retrieved.**

<u>Pseudocode</u>

```
Read BUTTON clicked by admin

ORDER = Order.get(id = ID of BUTTON)

CUSTOMER = Customer.get(id = ORDER_customer_id)

ADDRESS = ShippingAddress.get(id = ORDER_shippingaddress_id)

Output ORDER_id
Output ORDER_date_ordered
Output CUSTOMER_first_name, CUSTOMER_last_name
Output CUSTOMER_email
Output ADDRESS_address, ADDRESS_city, ADDRESS_state, ADDRESS_zipcode

Loop for ITEM in OrderItem.get(order_id = ORDER_id)
    PRODUCT_ID = ITEM_product_id            //As ITEM is a record of OrderItem
    PRODUCT = Product.get(id = PRODUCT_ID)
    Output PRODUCT_name, ITEM_quantity
End loop
```

```
Output TOTAL          //Calculated previously during customer checkout

Output ORDER_complete
Output ORDER_status
Output ORDER_payment_method
Output ORDER_paid
```



Updating Orders

If certain details such as the payment or order statuses of an order are updated, the **administrator** should be able to **reflect this** to the **user.** For this, the **administrator** should be able to **update fields such as:**

- Order status, if the order is dispatched/delivered.
- Paid, if the order has been paid for after delivery

If the admin then changes the value of either of these fields, they will then have to confirm these changes **using a 'Confirm' button**.

The logic for this presentation and updating would be as follows:

Pseudocode

```
If clickOn("Change order status" button) then
        Read NEW_STATUS

        If clickOn("Confirm" button) then
            ORDER_status = NEW_STATUS
        End if

Else if clickOn("Change payment status" button) then
        Read NEW_PAID_STATUS

        If clickOn("Confirm" button) then
            ORDER_paid = NEW_PAID_STATUS
End if
```

Filtering Orders

To make the process of **checking and managing orders easier** for the **administrator,** I will **allow them to filter** through **orders** on **parameters** such as:

- The payment method
- The payment status (whether 'paid' boolean is true or false)
- The order status

To do so, the logic would be as shown below. In this logic, I am assuming that pre-defined filter names will be given to the admin; from which they can select the one they wish to filter the orders list based on. These pre-defined inputs will be:

Filter	Options
"Payment method"	"Razorpay", "Cash on Delivery"
"Payment status"	True, False
"Order status"	"Received", "Dispatched", "Delivered"

Pseudocode

```
Method createOrderButton(ORDER)
    DATA = {ORDER_id,
            ORDER_customer_id,
            ORDER_date_ordered,
            ORDER_status,
            ORDER_payment_method,
            ORDER_paid}

    BUTTON = button.create(ID = ORDER_id, data = DATA)
    Return BUTTON
End method


Read FILTER

If FILTER == "Payment method" then
    Read OPTION
    For ORDER in Order.get(payment_method = OPTION)
        ORDER_BUTTON = createOrderButton(ORDER)
        Output ORDER_BUTTON
    End loop

Else if FILTER == “Payment status” then
    Read OPTION
    For ORDER in Order.get(paid = OPTION)
        ORDER_BUTTON = createOrderButton(ORDER)
        Output ORDER_BUTTON
    End loop

Else if FILTER == “Order status” then
    Read OPTION
    For ORDER in Order.get(status = OPTION)
        ORDER_BUTTON = createOrderButton(ORDER)
        Output ORDER_BUTTON
    End loop
End if
```

Sorting Orders

To further help **streamline** the process of **searching through/managing orders,** I will give the **administrators** the ability to **sort through orders.** The **parameters** I will allow them to **sort based on** are:

- Order ID
- Date ordered

The logic for this will be as follows (selection sort used given better time complexity)

Pseudocode

```
//Let ORDERS be an array of all the orders placed
ORDERS = Order.get.all()     //Retrieves all order IDs

MIN = 0
Read PARAMETER   //ID or Date

If PARAMETER == “ID” then
    Loop for I from 0 to ORDERS.length – 1
        MIN = I
        Loop for J from I+1 to ORDERS.length – 1
            If ORDERS[J]_id < ORDERS[MIN]_id then
                MIN = J
            End if
        End loop

        TEMP = ORDERS[I]
        ORDERS[I] = ORDERS[MIN]
        ORDERS[MIN] = TEMP
    End loop

Else if PARAMETER == “Date” then
    Loop for I from 0 to ORDERS.length – 1
        MIN = I
        Loop for J from I+1 to ORDERS.length – 1
            If ORDERS[J]_date_ordered < ORDERS[MIN]_date_ordered then
                MIN = J
            End if
        End loop

        TEMP = ORDERS[I]
        ORDERS[I] = ORDERS[MIN]
        ORDERS[MIN] = TEMP
    End loop
End if
```

Frontend Design Ideas

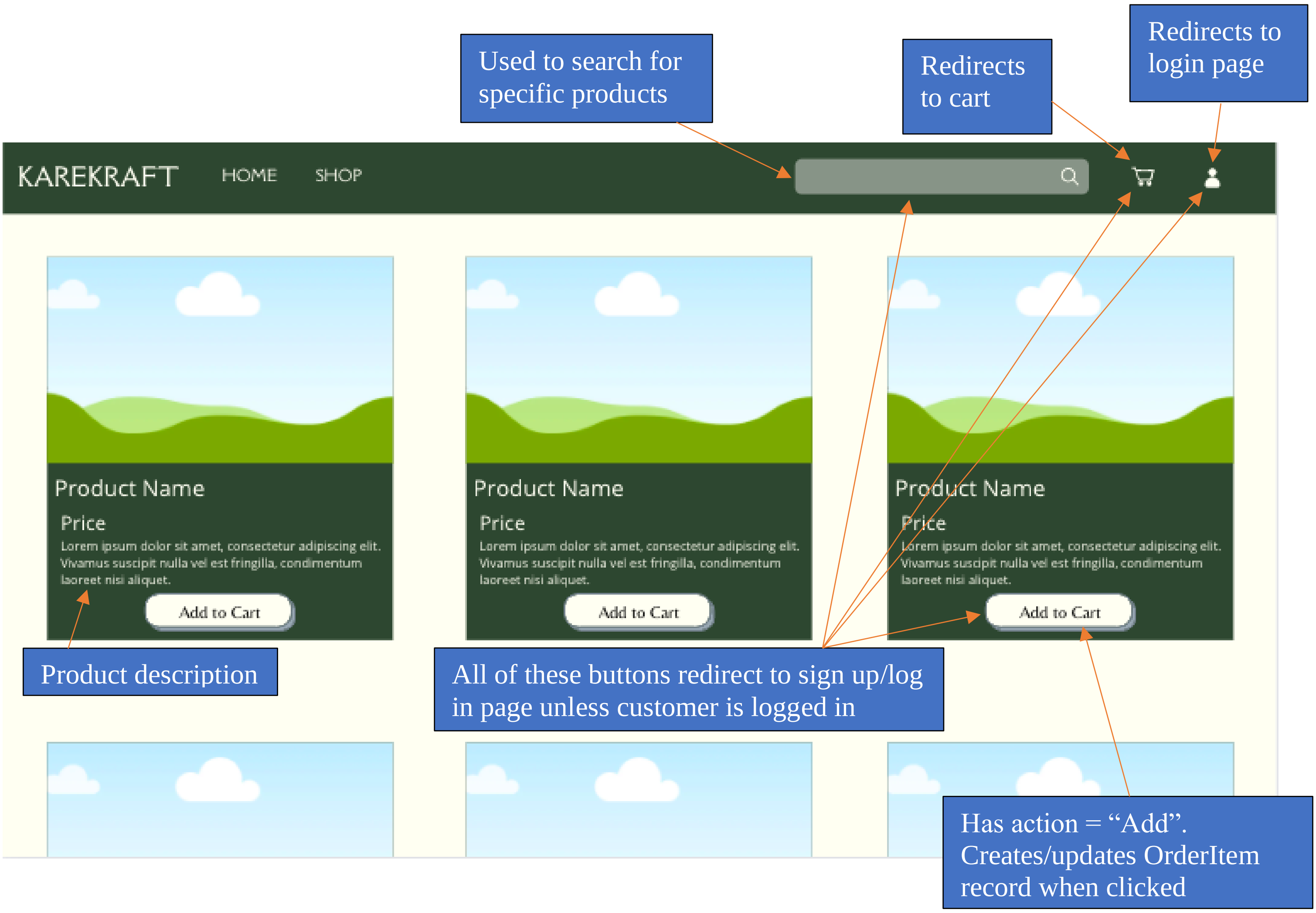
Customer Site Designs

Homepage

As the homepage is the main landing page of the website and will be used to display all the products hosted by my client, I found it necessary to ideate several designs for this page and take my client's input to finalize on one.

Design Idea 1

Within this design, all the products will be listed in rows and columns on one long, scrollable page, without any divisions based on category. The colour palette has been chosen basis my **client's brand colours.**


KAREKRAFT
HOME
SHOP
Product Name
Price
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus suscipit nulla vel est fringilla, condimentum laoreet nisi aliquet.
Add to Cart
Product Name
Price
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus suscipit nulla vel est fringilla, condimentum laoreet nisi aliquet.
Add to Cart
Product Name
Price
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus suscipit nulla vel est fringilla, condimentum laoreet nisi aliquet.
Add to Cart

Used to search for specific products

Redirects to cart

Redirects to login page

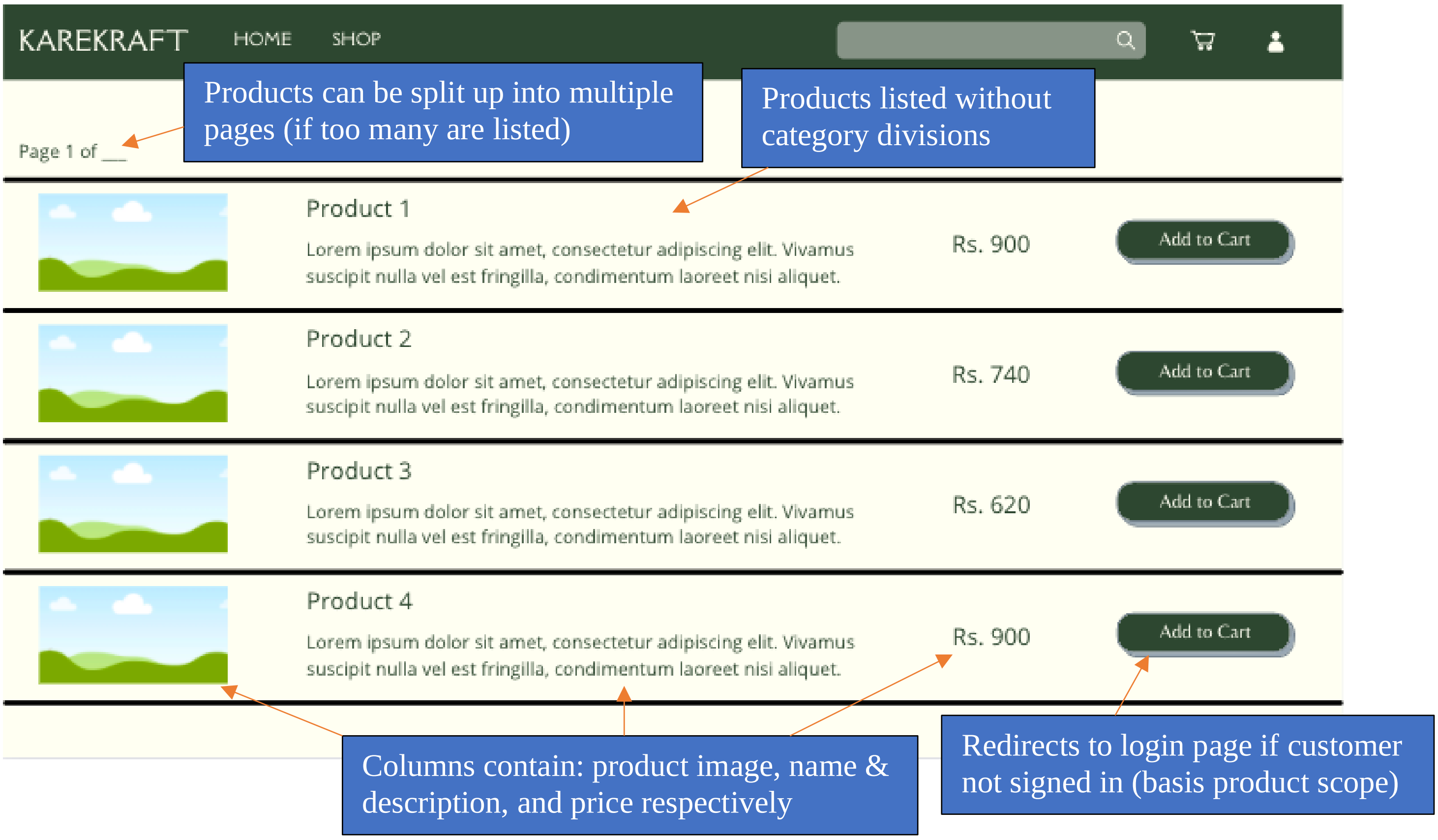
Product description

All of these buttons redirect to sign up/log in page unless customer is logged in

Has action = "Add". Creates/updates OrderItem record when clicked

Design Idea 2

Within this design, all the products are yet again listed without any division basis category, but are now listed as rows rather than individual box elements. This gives the website a similar aesthetic to e-commerce sites like Amazon.



Products can be split up into multiple pages (if too many are listed)

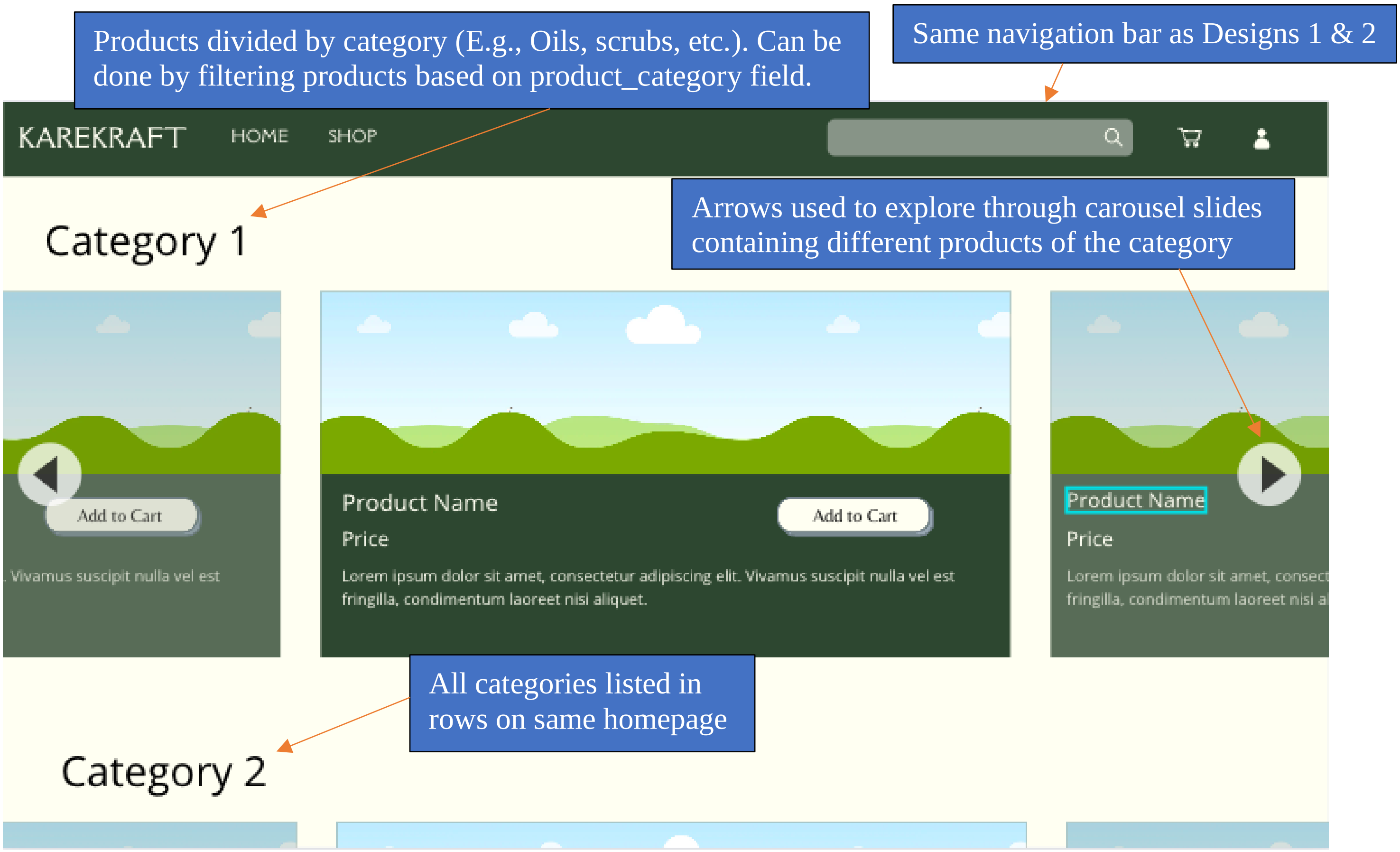
Products listed without category divisions

Columns contain: product image, name & description, and price respectively

Redirects to login page if customer not signed in (basis product scope)

Design Idea 3

Within this design, products are segregated in rows basis their category (E.g., oils, masks, serums etc.). Carousel slides can then be used to explore different products within a category.



KAREKRAFT
HOME
SHOP
Category 1
Add to Cart
Product Name
Price
Add to Cart
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus suscipit nulla vel est fringilla, condimentum laoreet nisi aliquet.
Product Name
Price
Category 2

Products divided by category (E.g., Oils, scrubs, etc.). Can be done by filtering products based on product_category field.

Same navigation bar as Designs 1 & 2

Arrows used to explore through carousel slides containing different products of the category

All categories listed in rows on same homepage

Having prepared these designs, I then sat down with my client and evaluated each design basis the relevant success criteria. Overall, the client believed that the following baseline success criteria was met by all the designs:

1. All details about the products available in stock are displayed to the customer.

2. Customers should be able to view products basis their category, or search for specific products by name.

3. There should be a navigation bar that allows customers to move between all the pages seamlessly.

Additionally, the client also gave their own functional and aesthetic evaluation of the strengths and limitations of each design basis the product scope and their personal opinions[8].

	Strengths	**Limitations**	**Suggested Improvements**
Design 1	Box element design of displaying products is: 1. Aesthetically pleasing, as it provides a concise summary of each product. 2. Allows all products to be listed on a single page, not requiring too much scrolling/navigation. 3. Use of multiple boxes in each row makes good use of the page, it doesn't feel empty 4. Contrasting box colours help products stand out	The lack of division of products basis category must be remedied, as it makes products easier to locate.	To retain the box element design but split page basis categories (such as in Design 3), or create individual pages for each category.
Design 2	1. Row-based product listing makes good use of the page space. 2. Client felt that the discrete columns made for the image, description, and price gave a good sense of organization.	1. Listing all products in long rows is not aesthetically pleasing, makes the page look monotonous. 2. Segregation of products basis category is necessary to making products easier to locate	While the products could be segregated based on category, retaining this row-based listing wouldn't be preferred by the client.
Design 3	1. Carousel slides design is quite aesthetically pleasing and adds to the sophistication of the website	While the slides are aesthetically pleasing, they reduce usability, as the customer must scroll through multiple other products to locate the one they wish to	Merging designs 1 & 3, i.e., keeping the category-based divisions from Design 3 and replacing the carousel slides

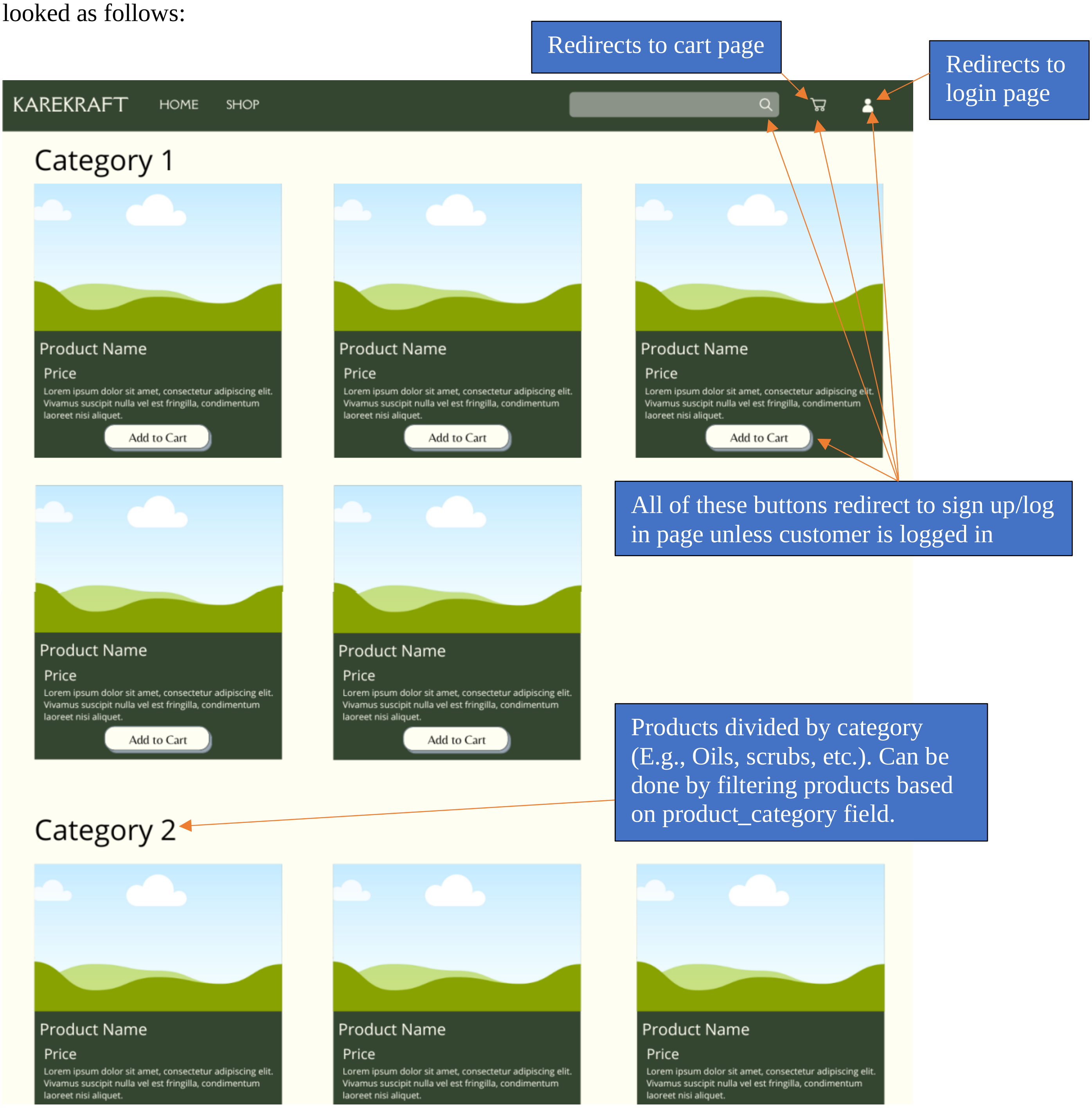
[8] Refer to Appendix – 'Criteria B: Evaluating Website Designs'

	2. Similar to the box elements, the contrasting box colour makes the products stand out 3. Division of products basis category helps with making it easier to locate specific products.	buy. As per the client, this difficult navigation process would personally deter them from perusing through too many products.	with the box elements from Design 1.

Following this, I also evaluated these designs from the viewpoint of a developer, basis which I had the following feedback:

	Strengths	**Limitations**
Design 1	• Contrast between box element and background colours improve usability/ readability by making text easy to read • Individual box elements for each product increases usability, as locating specific products becomes easier	• Category-based segregation of products would improve usability + help meet the client's success criteria
Design 2	• Each row acts as a discrete product summary, which makes comparison between products easier	• Usability is reduced as locating a specific product/category from the list becomes more difficult as the number of products increases (unless using search function). • Closely packed rows reduce usability as well, as it overloads the customer with too much densely-packed information.
Design 3	• Contrast between slide and background colours improve usability/readability by making text easy to read. • Category-based division increases usability by clustering products into groups, hence making browsing easier for customers.	• Implementing category-based division of products would **increase loading time of homepage** (as the products table would have to be iterated through multiple times to query and output each category's products). However, as this is an essential success criteria indicated by the client, it must be met. • Carousel slides reduce usability as it is more difficult to navigate through products within a category + more time-consuming to locate a specific product. • Comparison between products becomes difficult for customers as they cannot view multiple products together

Basis the client feedback and my own evaluation, I decided to merge designs 1 and 3 (as suggested by my client) to arrive at the final design. After making these changes, the design looked as follows:



Redirects to cart page

Redirects to login page

All of these buttons redirect to sign up/log in page unless customer is logged in

Products divided by category (E.g., Oils, scrubs, etc.). Can be done by filtering products based on product_category field.

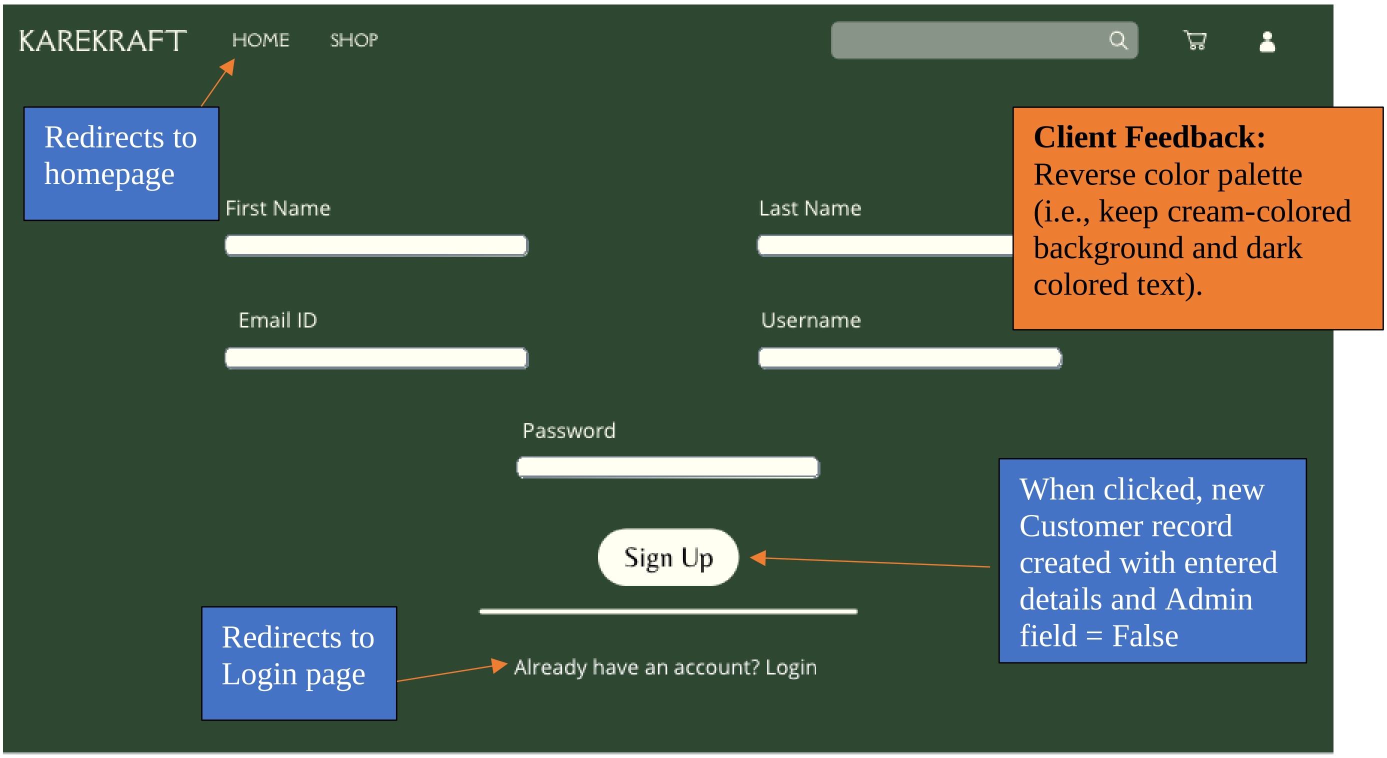
KAREKRAFT
HOME
SHOP
Category 1
Product Name
Price
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus suscipit nulla vel est fringilla, condimentum laoreet nisi aliquet.
Add to Cart
Category 2

As seen, this design merges the category-based division of products included in Design idea 3, and merges it with the box-element product listing that the client preferred from Design 1. Furthermore, having shown this final design to my client, it was approved by her, as she felt it met all her functional and aesthetic requirements, as well as the success criteria[9].

[9] Refer to Appendix – 'Criteria B: Finalizing Homepage'

The following pages were also shown and approved by my client, with a few changes outlined for certain pages which I noted and will incorporate during final development[10].

Sign Up Page



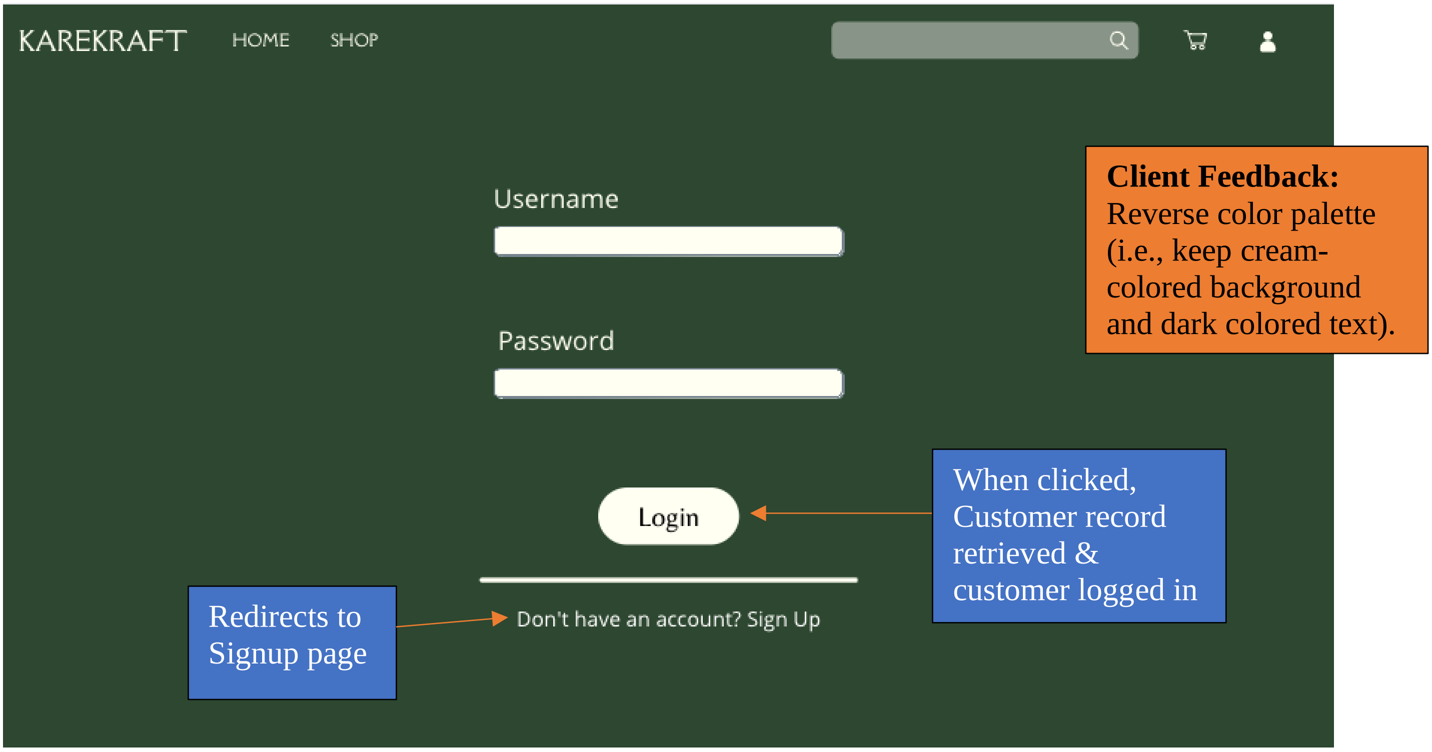
Redirects to homepage

Client Feedback: Reverse color palette (i.e., keep cream-colored background and dark colored text).

When clicked, new Customer record created with entered details and Admin field = False

Redirects to Login page

Login Page



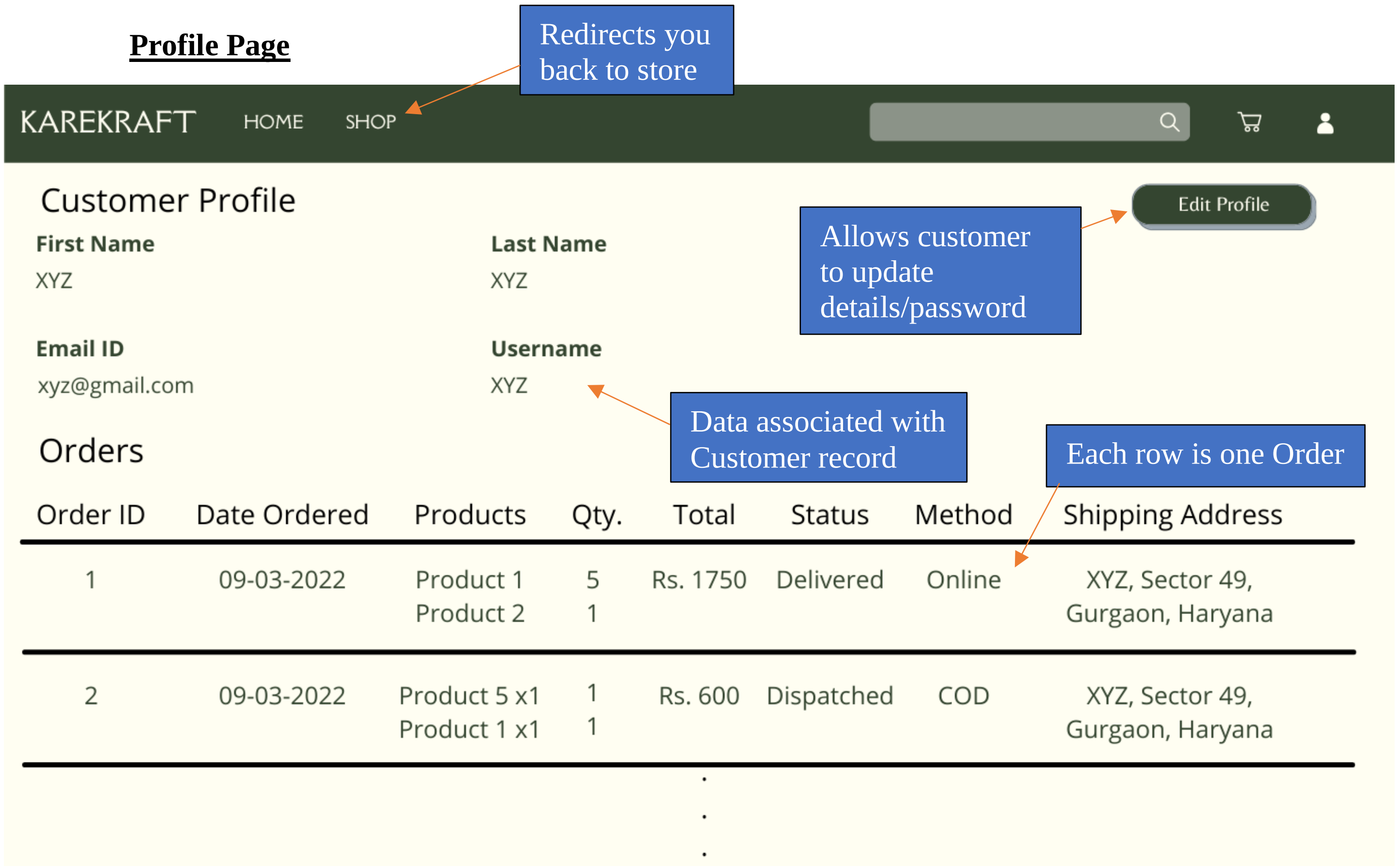
Client Feedback: Reverse color palette (i.e., keep cream-colored background and dark colored text).

When clicked, Customer record retrieved & customer logged in

Redirects to Signup page

[10] Refer to Appendix – 'Criteria B: Evaluating Website Designs'

Profile Page



Redirects you back to store

Customer Profile

Edit Profile

Allows customer to update details/password

First Name
XYZ

Last Name
XYZ

Email ID
xyz@gmail.com

Username
XYZ

Data associated with Customer record

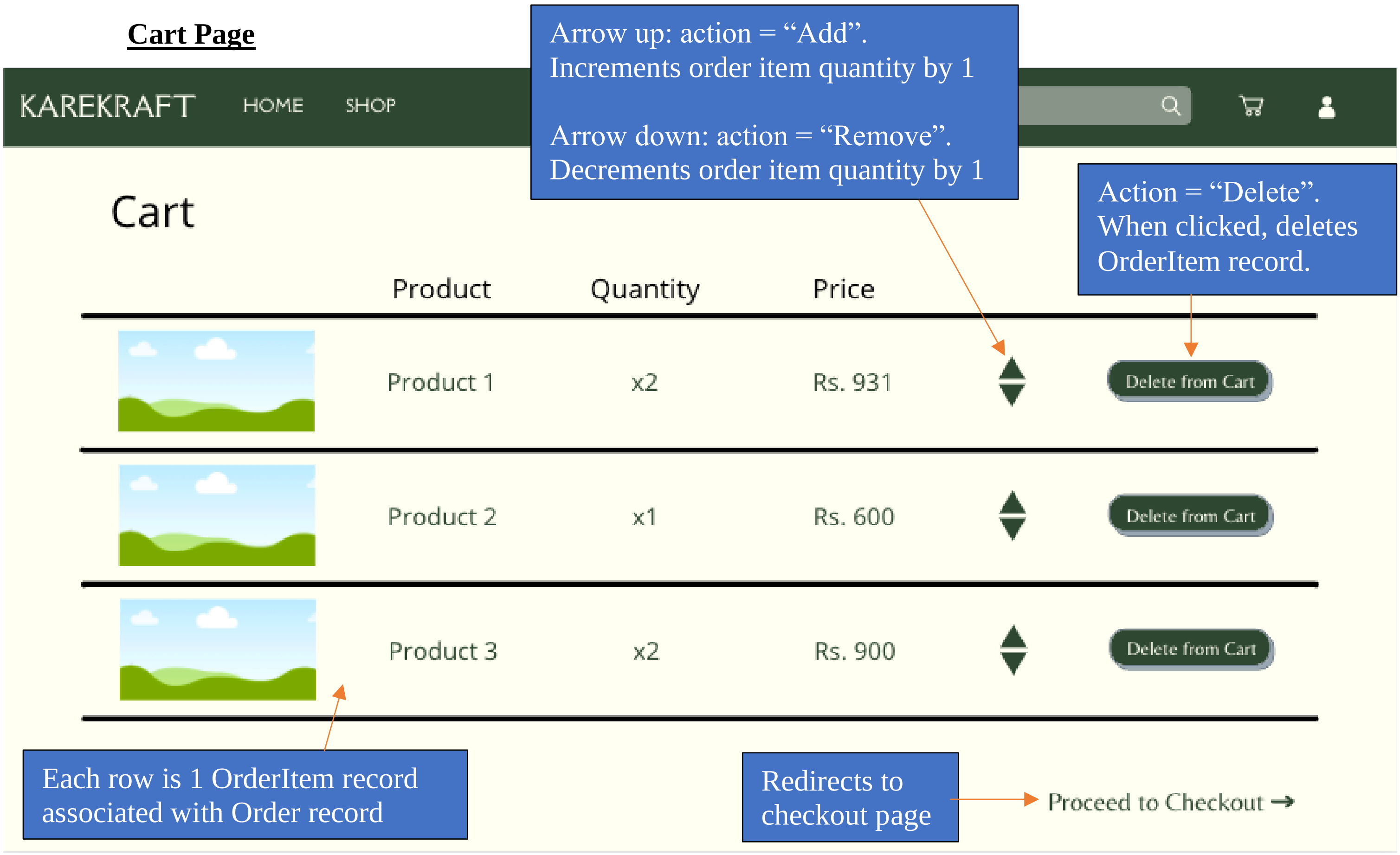
Orders

Each row is one Order

Order ID	Date Ordered	Products	Qty.	Total	Status	Method	Shipping Address
1	09-03-2022	Product 1 Product 2	5 1	Rs. 1750	Delivered	Online	XYZ, Sector 49, Gurgaon, Haryana
2	09-03-2022	Product 5 x1 Product 1 x1	1 1	Rs. 600	Dispatched	COD	XYZ, Sector 49, Gurgaon, Haryana

.
.
.

Cart Page



Arrow up: action = “Add”.
Increments order item quantity by 1

Arrow down: action = “Remove”.
Decrements order item quantity by 1

Cart

Action = “Delete”.
When clicked, deletes OrderItem record.

	Product	Quantity	Price		
	Product 1	x2	Rs. 931	▲▼	Delete from Cart
	Product 2	x1	Rs. 600	▲▼	Delete from Cart
	Product 3	x2	Rs. 900	▲▼	Delete from Cart

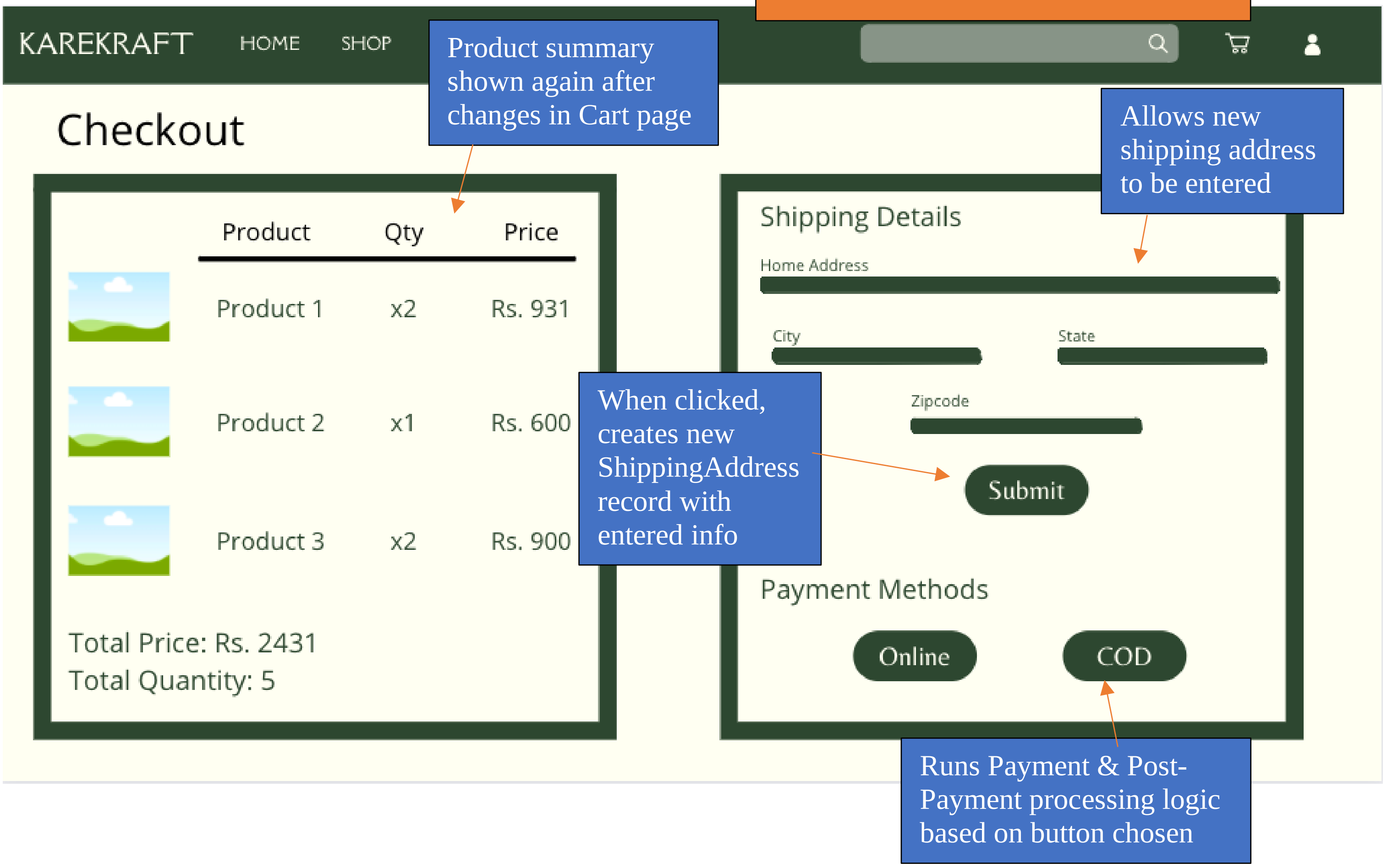
Each row is 1 OrderItem record associated with Order record

Redirects to checkout page

Proceed to Checkout →

Checkout Page

Client Feedback: Maximize use of space (keep elements spaced out to prevent overcrowding of information) + Reduce border width of columns



KAREKRAFT HOME SHOP

Checkout

	Product	Qty	Price
	Product 1	x2	Rs. 931
	Product 2	x1	Rs. 600
	Product 3	x2	Rs. 900

Total Price: Rs. 2431

Total Quantity: 5

Shipping Details

Home Address

City

State

Zipcode

Submit

Payment Methods

Online

COD

Product summary shown again after changes in Cart page

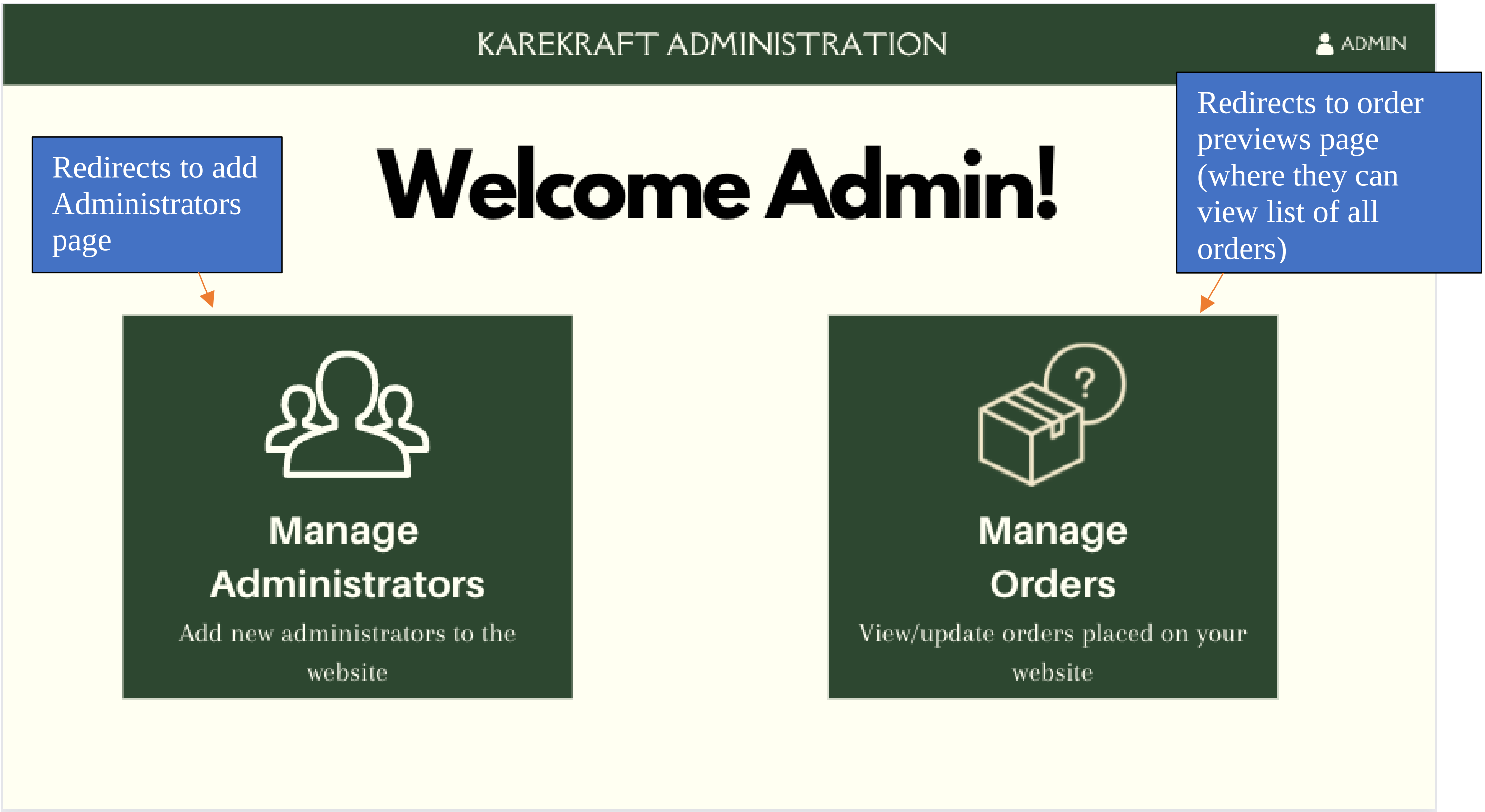
Allows new shipping address to be entered

When clicked, creates new ShippingAddress record with entered info

Runs Payment & Post-Payment processing logic based on button chosen

Admin Site Designs

Homepage



KAREKRAFT ADMINISTRATION

ADMIN

Welcome Admin!

Manage Administrators

Add new administrators to the website

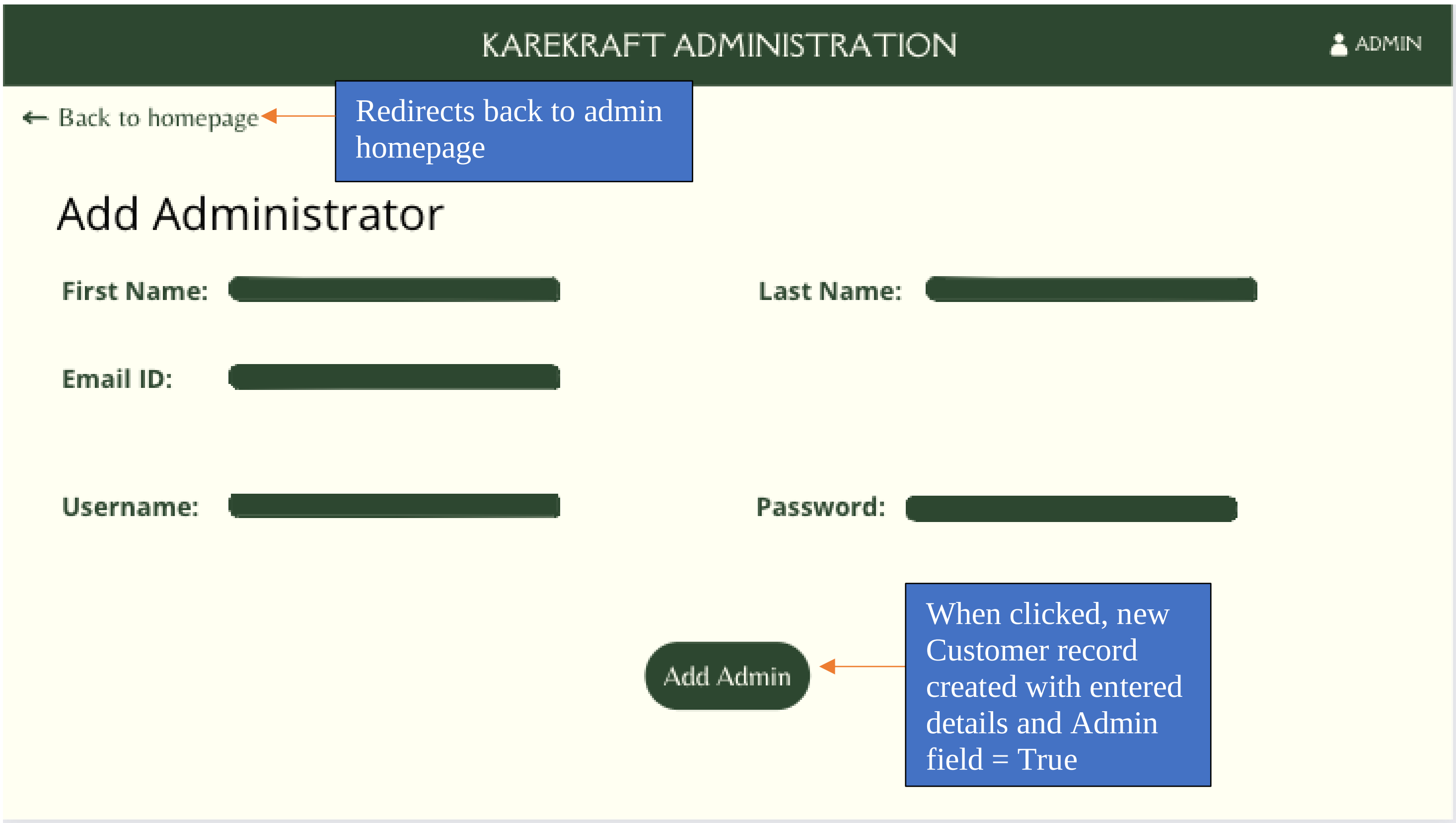
Manage Orders

View/update orders placed on your website

Redirects to add Administrators page

Redirects to order previews page (where they can view list of all orders)

Add Administrators Page



KAREKRAFT ADMINISTRATION

← Back to homepage

Redirects back to admin homepage

Add Administrator

First Name:

Last Name:

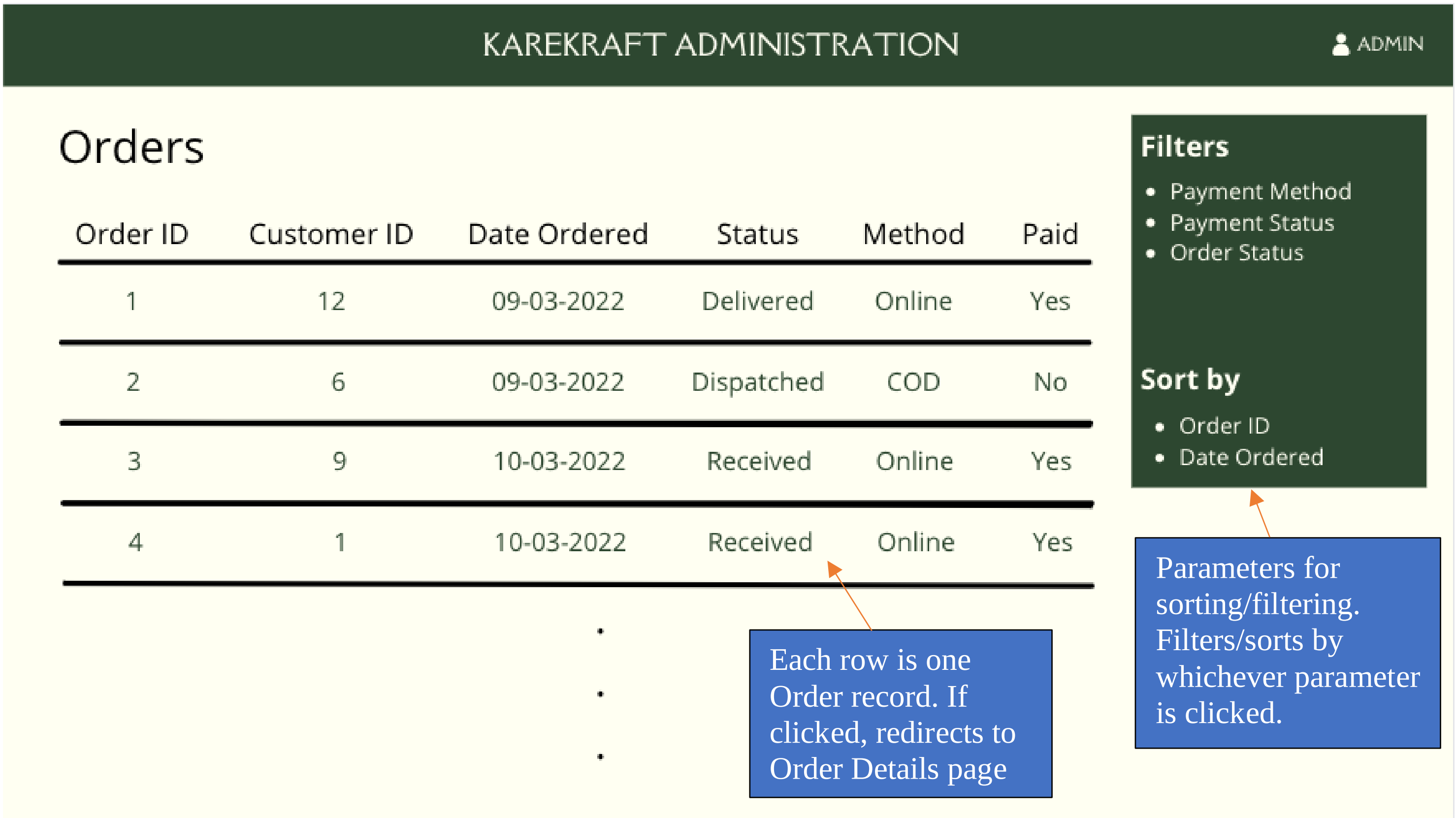
Email ID:

Username:

Password:

When clicked, new Customer record created with entered details and Admin field = True

Order Previews Page



KAREKRAFT ADMINISTRATION

Orders

Order ID	Customer ID	Date Ordered	Status	Method	Paid
1	12	09-03-2022	Delivered	Online	Yes
2	6	09-03-2022	Dispatched	COD	No
3	9	10-03-2022	Received	Online	Yes
4	1	10-03-2022	Received	Online	Yes

.

.

.

Filters

- Payment Method
- Payment Status
- Order Status

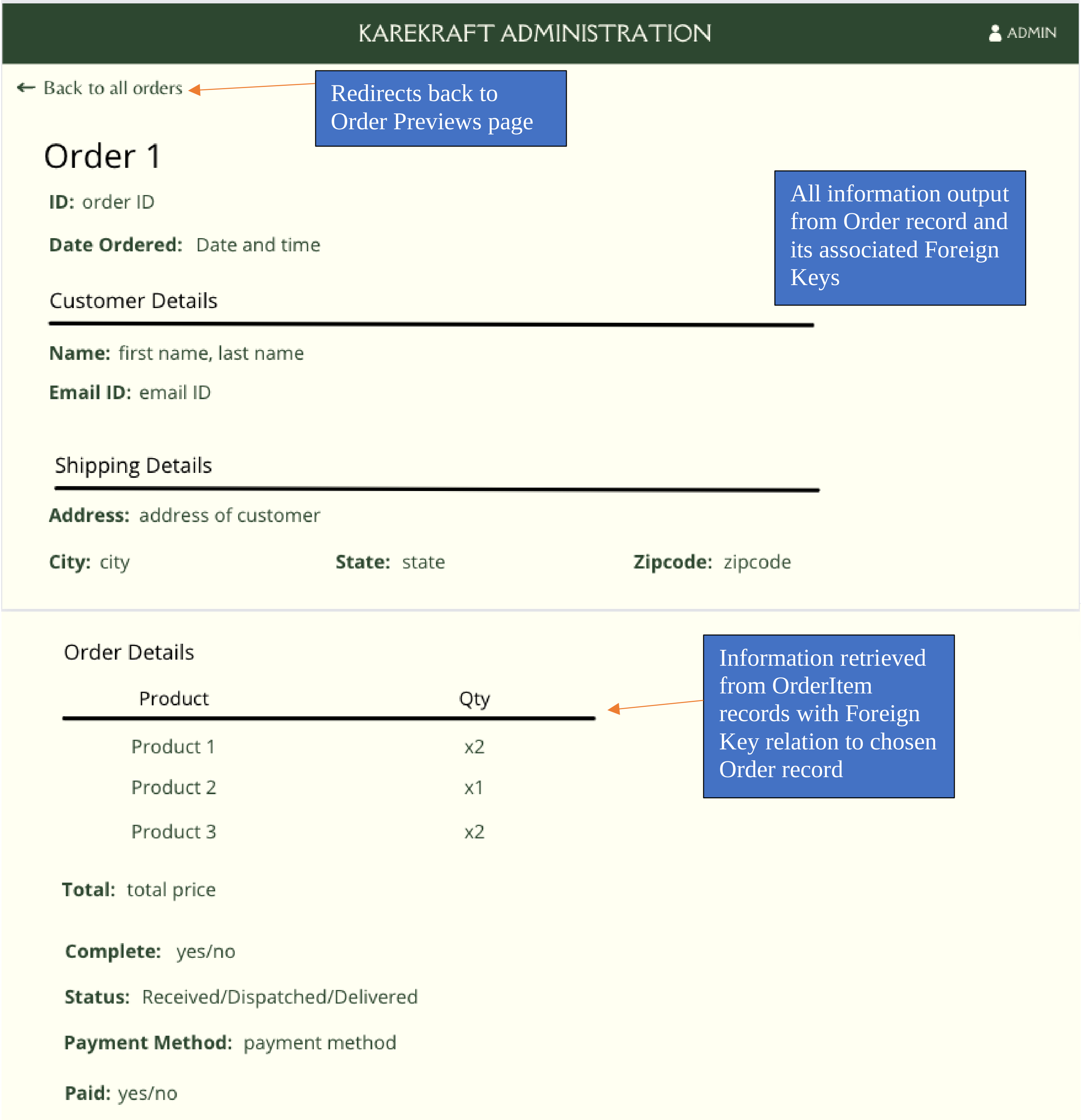
Sort by

- Order ID
- Date Ordered

Each row is one Order record. If clicked, redirects to Order Details page

Parameters for sorting/filtering. Filters/sorts by whichever parameter is clicked.

Order Details Page



KAREKRAFT ADMINISTRATION

ADMIN

← Back to all orders

Redirects back to Order Previews page

Order 1

ID: order ID

Date Ordered: Date and time

All information output from Order record and its associated Foreign Keys

Customer Details

Name: first name, last name

Email ID: email ID

Shipping Details

Address: address of customer

City: city **State:** state **Zipcode:** zipcode

Order Details

Product	Qty
Product 1	x2
Product 2	x1
Product 3	x2

Information retrieved from OrderItem records with Foreign Key relation to chosen Order record

Total: total price

Complete: yes/no

Status: Received/Dispatched/Delivered

Payment Method: payment method

Paid: yes/no

Test Plan

Customer-Side Functionality

Test Type	Nature of Test	Expected Result
Signup form	Relevant output in case of data entry error.	Error messages like: “Invalid email ID” outputted based on error in form.
	Returning an error if Customer record already exists.	Message like, “Account with this information already exists,” outputted.
	New Customer record created + customer logged-in once form is successfully submitted.	Customer’s account details reflect on profile page, and customer logged in.
Login form	Relevant output in case of data entry error.	Error messages like: “Invalid username” outputted based on error in form.
	Logging customer in if Customer record found in database.	Customer logged in & success message like “Logged in!” displayed.
Profile Page	All account details and previous order data visible to customer.	Username, contact information, and previous order data displayed.
	Customer can update account details.	Updated customer account details reflected in database and profile page.
Products Functionality	Adding/Deleting products to/from the customer’s cart.	Products added/deleted from list on Cart page, and relevant OrderItem record inserted/deleted from database table.
	Incrementing/Decrementing quantity of products in customer’s cart.	Product quantity updated in list on Cart page, and quantity field of relevant OrderItem record increased/reduced by 1.
Cart & Checkout Page	Correct output of all order data.	Each product, its price, and its quantity are output, alongside total price of order.
Shipping Addresses	Previous shipping addresses retrieved from ShippingAddress table and displayed.	Separate buttons for each saved shipping address are displayed.
	New shipping address added successfully.	New record associated with Customer’s ID created in ShippingAddress table with customer’s entered data.

	Correct ShippingAddress record retrieved if previously saved address selected.	Selected shipping address is displayed to customer after successful order placement.
Online payment	Payment request successfully sent to Razorpay.	Razorpay payment portal opens on website.
	If payment declines, error message shown to customer.	Outputs like, “Incorrect card details,” displayed.
	Verifying successful Razorpay payment.	Message like, “Payment successful,” and variables like Razorpay signature are not null.
Post-Payment	Relevant fields of Order record updated.	Order ‘status’, ‘paid’, ‘payment method’, shipping address ID, and ‘complete’ fields updated basis chosen payment method.
	Order successfully placed.	Success message displayed, customer redirected to homepage, and order reflects on profile page.
	Decrementing of product stocks.	Available stocks of each product decrement basis the quantity ordered.

Admin-Side Functionality

Test Type	Nature of Test	Expected Result
Admin Signup & Login	Refer to sign up & login described for customer	
Adding/Removing Products	Adding/Deleting/Updating products for the website.	New products successfully added, and existing products successfully deleted or have data (E.g., stock) updated.
Viewing Orders	Relevant fields of all Orders displayed in list.	Order ID, date ordered, order status, and paid fields displayed for each order row summary.
	All order details of a selected order displayed.	Relevant information about order items, customer details, and shipping address displayed on new page.
Updating Orders	Order status/payment status successfully updated if changed by admin.	Order ‘status’ and ‘paid’ fields for relevant Order record updated on customer’s profile page & database.

Filtering & Sorting Orders	Order rows sorted/filtered based on chosen parameter.	Only rows of Order table with relevant filter/sort parameter outputted.

Word Count: 495 words (excluding all headers)